

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

大数据的本质

大数据现状及挑战 驱动因素 未来趋势 Spark原理及应用

探针 爬虫 日志采集 Flink 深度学习 数据分发中间件

Broadview[®]
www.broadview.com.cn



大数据架构详解

从数据获取到深度学习

朱洁 罗华霖 / 编著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

作者介绍 ●●●●

朱洁

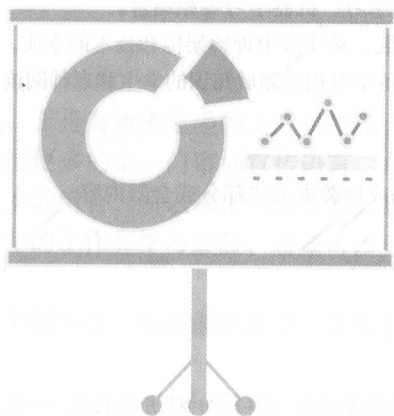


2008年加入华为，具有8年大数据研发管理经验，现任华为大数据服务首席规划师。专注于大数据服务平台建设、规划和实践应用，同时参与多项企业级大数据项目解决方案的规划、设计和实施工作，在深化大数据行业落地方面有诸多实践经验，对解读大数据垂直行业的技术创新与开发有诸多独到的见解和心得。

罗华霖



2002年加入华为，华为大数据首席规划师，主导完成华为大数据平台DataSight和华为电信大数据解决方案SmartCare的技术规划和架构设计，支持电信运营商数字化战略转型，完成浙江移动、上海联通、沙特STC等200+电信大数据解决方案项目落地。曾任华为软交换首席设计师、华为大型电信大数据解决方案SmartCare首席架构师。



大数据架构详解

从数据获取到深度学习

朱洁 罗华霖 / 编著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书从架构、业务、技术三个维度深入浅出地介绍了大数据处理领域端到端的知识。主要内容包括三部分：第一部分从数据的产生、采集、计算、存储、消费端到端的角度介绍大数据技术的起源、发展、关键技术点和未来趋势，结合生动的业界最新产品，以及学术界最新的研究方向和成果，让深奥的技术浅显易懂；第二部分从业务和技术角度介绍实际案例，让读者理解大数据的用途及技术的本质；第三部分介绍大数据技术不是孤立的，讲解如何与前沿的云技术、深度学习、机器学习等相结合。

本书内容深入浅出，技术结合实践，从实践中理解架构和技术的本质，适合大数据技术领域的从业人员如架构师、工程师、产品经理等，以及准备学习相关领域知识的学生和老师阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

大数据架构详解：从数据获取到深度学习 / 朱洁，罗华霖编著. —北京：电子工业出版社，2016.10

ISBN 978-7-121-30000-4

I. ①大… II. ①朱… ②罗… III. ①数据处理—研究 IV. ①TP274

中国版本图书馆 CIP 数据核字（2016）第 233622 号

策划编辑：安 娜

责任编辑：徐津平

特约编辑：赵树刚

印 刷：北京中新伟业印刷有限公司

装 订：北京中新伟业印刷有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×980 1/16 印张：23.25 字数：566 千字

版 次：2016 年 10 月第 1 版

印 次：2016 年 10 月第 1 次印刷

印 数：3000 册 定价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

前言

大数据这几年真的很火，于是有越来越多的人开始学习大数据技术。很多人会误以为大数据是一门技术，其实不然，大数据更多的是一门市场宣传语言，也可以理解为一种思考方式。从技术角度来看，大数据是一系列技术的组合，所以真正全面掌握大数据技术也是一件很困难的事情。编写这本书的初衷就是总结这些年的工作和学习经验，希望可以分享给更多人，同时对自己而言也是一个提高、总结和升华的过程。

总的来说，本书围绕一个通用技术栈来组织章节，主要聚焦大数据平台的一些知识。主要分为三部分。

第一部分：第 1~3 章，主要讲述大数据的本质、运营商大数据的架构和一些基本的业务知识。

- 第 1 章：阐述大数据的本质和面临的挑战。
- 第 2 章：概述大数据架构及背后的驱动因素，以及未来发展的趋势。
- 第 3 章：介绍运营商领域的业务，让读者对大数据能做什么有一个直观的感受。

第二部分：第 4~11 章，围绕大数据平台技术栈来阐述数据获取、处理、分析和应用平台涉及的技术。

- 第 4 章：介绍数据获取涉及的探针、爬虫、日志采集、数据分发中间件等技术。
- 第 5 章：介绍流式数据处理引擎、CEP、流式应用。
- 第 6 章：介绍交互式分析技术、MPP DB、热门的 SQL on Hadoop 技术。
- 第 7 章：介绍批处理技术、Spark，以及大规模机器学习的 BSP 技术等。
- 第 8 章：探讨机器学习、深度学习相关技术。
- 第 9 章：统一资源管理是趋势，本章介绍资源管理的核心技术和算法。
- 第 10 章：存储是基础，本章介绍存储的关键技术。
- 第 11 章：探讨大数据技术怎么云化，以及关键技术是什么。

第三部分：第 12 章，技术和文化息息相关，技术影响文化，文化影响技术。

第 12 章：介绍大数据开发文化、开源、DevOps，探讨理念和文化对技术的冲击。

由于编者水平有限，书中疏漏之处在所难免，敬请谅解。

最后以乔布斯的经典名句结尾：Stay hungry, Stay foolish。

朱 洁

2016 年 5 月于深圳

目 录

第一部分 大数据的本质

第 1 章 大数据是什么.....	2
1.1 大数据导论.....	2
1.1.1 大数据简史.....	2
1.1.2 大数据现状.....	3
1.1.3 大数据与 BI.....	3
1.2 企业数据资产.....	4
1.3 大数据挑战.....	5
1.3.1 成本挑战.....	6
1.3.2 实时性挑战.....	6
1.3.3 安全挑战.....	6
1.4 小结.....	6
第 2 章 运营商大数据架构.....	7
2.1 架构驱动的因素.....	7
2.2 大数据平台架构.....	7
2.3 平台发展趋势.....	8
2.4 小结.....	8
第 3 章 运营商大数据业务.....	9
3.1 运营商常见的大数据业务.....	9
3.1.1 SQM (运维质量管理).....	9
3.1.2 CSE (客户体验提升).....	9
3.1.3 MSS (市场运维支撑).....	10
3.1.4 DMP (数据管理平台).....	10
3.2 小结.....	11

第二部分 大数据技术

第 4 章 数据获取	14
4.1 数据分类	14
4.2 数据获取组件	14
4.3 探针	15
4.3.1 探针原理	15
4.3.2 探针的关键能力	16
4.4 网页采集	26
4.4.1 网络爬虫	26
4.4.2 简单爬虫 Python 代码示例	32
4.5 日志收集	33
4.5.1 Flume	33
4.5.2 其他日志收集组件	47
4.6 数据分发中间件	47
4.6.1 数据分发中间件的作用	47
4.6.2 Kafka 架构和原理	47
4.7 小结	82
第 5 章 流处理	83
5.1 算子	83
5.2 流的概念	83
5.3 流的应用场景	84
5.3.1 金融领域	84
5.3.2 电信领域	85
5.4 业界两种典型的流引擎	85
5.4.1 Storm	85
5.4.2 Spark Streaming	89
5.4.3 融合框架	102
5.5 CEP	108
5.5.1 CEP 是什么	108
5.5.2 CEP 的架构	109
5.5.3 Esper	110

5.6 实时结合机器学习	110
5.6.1 Eagle 的特点	111
5.6.2 Eagle 概览	111
5.7 小结	116
第 6 章 交互式分析	117
6.1 交互式分析的概念	117
6.2 MPP DB 技术	118
6.2.1 MPP 的概念	118
6.2.2 典型的 MPP 数据库	121
6.2.3 MPP DB 调优实战	131
6.2.4 MPP DB 适用场景	162
6.3 SQL on Hadoop	163
6.3.1 Hive	163
6.3.2 Phoenix	165
6.3.3 Impala	166
6.4 大数据仓库	167
6.4.1 数据仓库的概念	167
6.4.2 OLTP/OLAP 对比	168
6.4.3 大数据场景下的同与不同	168
6.4.4 查询引擎	169
6.4.5 存储引擎	170
6.5 小结	171
第 7 章 批处理技术	172
7.1 批处理技术的概念	172
7.2 MPP DB 技术	172
7.3 MapReduce 编程框架	173
7.3.1 MapReduce 起源	173
7.3.2 MapReduce 原理	173
7.3.3 Shuffle	174
7.3.4 性能差的主要原因	177
7.4 Spark 架构和原理	177
7.4.1 Spark 的起源和特点	177

7.4.2	Spark 的核心概念	178
7.5	BSP 框架	217
7.5.1	什么是 BSP 模型	217
7.5.2	并行模型介绍	218
7.5.3	BSP 模型基本原理	220
7.5.4	BSP 模型的特点	222
7.5.5	BSP 模型的评价	222
7.5.6	BSP 与 MapReduce 对比	222
7.5.7	BSP 模型的实现	223
7.5.8	Apache Hama 简介	223
7.6	批处理关键技术	227
7.6.1	CodeGen	227
7.6.2	CPU 亲和技术	228
7.7	小结	229
第 8 章	机器学习和数据挖掘	230
8.1	机器学习和数据挖掘的联系与区别	230
8.2	典型的数据挖掘和机器学习过程	231
8.3	机器学习概览	232
8.3.1	学习方式	232
8.3.2	算法类似性	233
8.4	机器学习&数据挖掘应用案例	235
8.4.1	尿布和啤酒的故事	235
8.4.2	决策树用于电信领域故障快速定位	236
8.4.3	图像识别领域	236
8.4.4	自然语言识别	238
8.5	交互式分析	239
8.6	深度学习	240
8.6.1	深度学习概述	240
8.6.2	机器学习的背景	241
8.6.3	人脑视觉机理	242
8.6.4	关于特征	244
8.6.5	需要有多少个特征	245
8.6.6	深度学习的基本思想	246

8.6.7 浅层学习和深度学习	246
8.6.8 深度学习与神经网络	247
8.6.9 深度学习的训练过程	248
8.6.10 深度学习的框架	248
8.6.11 深度学习与 GPU	255
8.6.12 深度学习小结与展望	256
8.7 小结	257
第 9 章 资源管理	258
9.1 资源管理的基本概念	258
9.1.1 资源调度的目标和价值	258
9.1.2 资源调度的使用限制及难点	258
9.2 Hadoop 领域的资源调度框架	259
9.2.1 YARN	259
9.2.2 Borg	260
9.2.3 Omega	262
9.2.4 本节小结	263
9.3 资源分配算法	263
9.3.1 算法的作用	263
9.3.2 几种调度算法分析	263
9.4 数据中心统一资源调度	271
9.4.1 Mesos+Marathon 架构和原理	271
9.4.2 Mesos+Marathon 小结	283
9.5 多租户技术	284
9.5.1 多租户概念	284
9.5.2 多租户方案	284
9.6 基于应用描述的智能调度	287
9.7 Apache Mesos 架构和原理	288
9.7.1 Apache Mesos 背景	288
9.7.2 Apache Mesos 总体架构	288
9.7.3 Apache Mesos 工作原理	290
9.7.4 Apache Mesos 关键技术	295
9.7.5 Mesos 与 YARN 比较	304
9.8 小结	305

第 10 章 存储是基础	306
10.1 分久必合，合久必分	306
10.2 存储硬件的发展	306
10.2.1 机械硬盘的工作原理	306
10.2.2 SSD 的原理	307
10.2.3 3DXPoint	309
10.2.4 硬件发展小结	309
10.3 存储关键指标	309
10.4 RAID 技术	309
10.5 存储接口	310
10.5.1 文件接口	311
10.5.2 裸设备	311
10.5.3 对象接口	312
10.5.4 块接口	316
10.5.5 融合是趋势	328
10.6 存储加速技术	328
10.6.1 数据组织技术	328
10.6.2 缓存技术	335
10.7 小结	336
第 11 章 大数据云化	337
11.1 云计算定义	337
11.2 应用上云	337
11.2.1 Cloud Native 概念	338
11.2.2 微服务架构	338
11.2.3 Docker 配合微服务架构	342
11.2.4 应用上云小结	348
11.3 大数据上云	348
11.3.1 大数据云服务的两种模式	348
11.3.2 集群模式 AWSEMR	349
11.3.3 服务模式 Azure Data Lake Analytics	352
11.4 小结	354

第三部分 大数据文化

第 12 章 大数据技术开发文化.....	356
12.1 开源文化.....	356
12.2 DevOps 理念.....	356
12.2.1 Development 和 Operations 的组合	357
12.2.2 对应用程序发布的影响	357
12.2.3 遇到的问题.....	358
12.2.4 协调人.....	358
12.2.5 成功的关键.....	359
12.3 速度远比你想要的重要.....	359
12.4 小结.....	361

第一部分 大数据的本质

第 1 章

大数据是什么

首先提一个问题：“大数据”是一项专门的技术吗？有的人可能会以为大数据是一项专门的技术，其实不是。“大数据”这三个字只是一门市场语言（Marketing Language），其背后是硬件、数据库、操作系统、Hadoop 等一系列技术的综合应用，所以本书我们希望从一个端到端的架构展开讲解典型的大数据技术。

1.1 大数据导论

1.1.1 大数据简史^①

大数据（Big Data）现在可以说是人尽皆知，其实真正回溯起来，其是由 SGI 的首席科学家 John R. Masey 于 1998 年在 USENIX 大会上首次提出的。他在其发表的一篇名为 *Big Data and the Next Wave of Infrastrress*^② 的论文中首次提出这个词，用来描述数据爆炸的现象。估计他当时未必能想到十几年后 Big Data 能这么火。

如果追溯大数据的概念，则是阿尔文·托夫勒（Alvin Toffler）于 1980 年在《第三次浪潮》一书中预言了信息时代的到来会带来数据爆发，所以科学家很早就预见到了大数据。大数据的历史由来已久，但是技术需要持续积累，才能由量变到质变。

对于工业界来说，不得不提 Google 在 2003—2006 年公布的关于 GFS、MapReduce 和 BigTable 的三篇技术论文，正是这三篇论文奠定了大数据发展的基石。Hadoop 之父——Doug Cutting^③ 正是参考论文，后来才实现了当前鼎鼎大名的 Hadoop，而 Hadoop 的诞生极大地促进了大数据技术的蓬勃发展。

当然，这里特别要指出，Hadoop 并不等同于大数据，大数据也并不特指 Hadoop，大数据只是一门市场语言，代表的是一种理念、一种问题解决思路、一系列技术的集合，Hadoop 只是其中一种具体的处理数据的框架技术。

① 参考 forbes <http://www.forbes.com/sites/gilpress/2013/05/09/a-very-short-history-of-big-data/>。

② *Big Data and the Next Wave of Infrastrress* (<https://www.usenix.org/conference/1999-usenix-annual-technical-conference/big-data-and-next-wave-infrastrress-problems>)。

③ Doug Cutting 是 Lucene、Nutch、Hadoop 等项目的发起人。

1.1.2 大数据现状

Gartner 发布的 2016 技术成熟度曲线 (见图 1.1) 首次将云计算、大数据及相关技术移除。Gartner 指出这些技术不是不重要, 而是不再“新兴”, 大家虽然对大数据的兴趣依然不减, 但是这个市场已经安定下来, 有了一整套合理的方法, 新的技术和实践被添加进现有方案。所以大数据度过了技术的期望膨胀高峰期, 到了真正使用大数据解决问题的时候。未来大数据相关技术的演进在很长一段时间仍将展现出强大的生命力, 相关市场的营收也将不断放大。

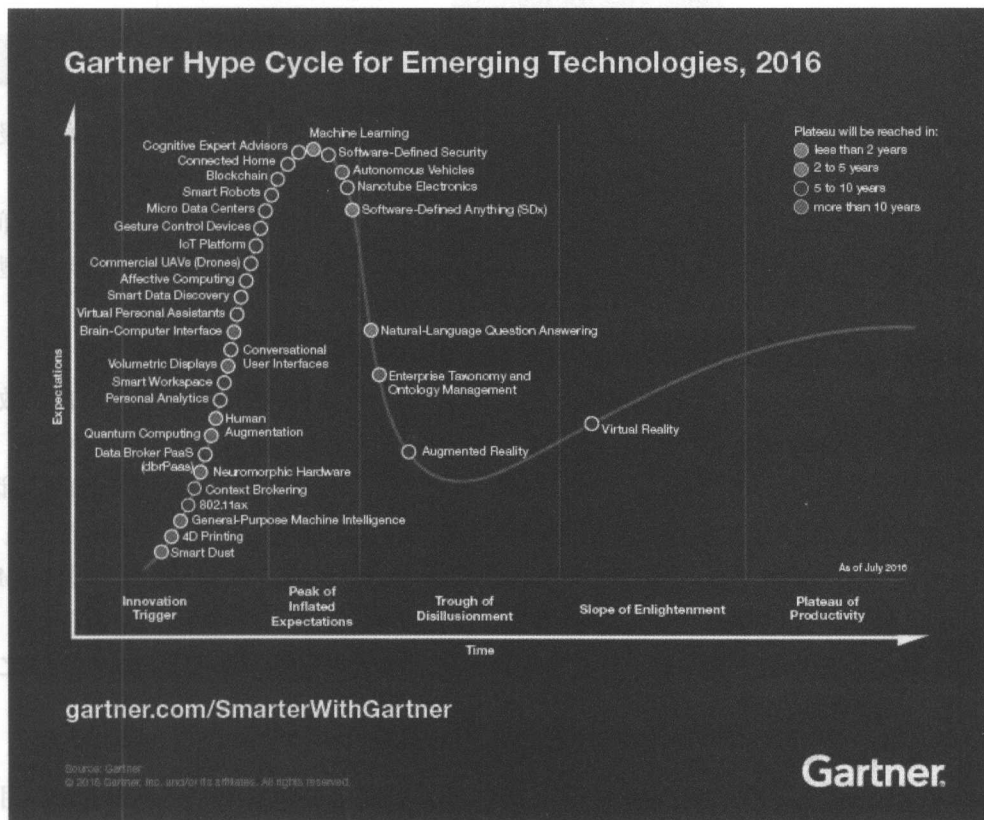


图 1.1

1.1.3 大数据与 BI^①

前面说了大数据是一种理念、一种问题解决思路 and 一系列技术的集合, 它与传统的 BI 既有相同之处, 也有不同之处。

相同之处, 都是从数据中挖掘价值, 促进商业成功。不同之处, 核心是分布式技术的发展、处理能力的极大提高, 以前想都不敢想的处理变成了可能。所以在对数据的处理理念上也得到了扩展:

^① Business Intelligence, 商业智能。

(1) 不局限于传统的 BI 从数据中抽样建模，再回 DW^①实施，大数据可以直接从全量数据中找出规律，通过数据的样本多样化弥补模型的准确性。

(2) 不局限于传统的 BI 简单地通过汇总、统计分析找出群体共性从而输出报表，大数据可以直接通过足够多的数据对个体进行刻画。

虽然有种种不同，但未来大数据和 BI 的界限会模糊，企业的核心驱动目标是从数据资产中找出商业价值，而不关心构建和分析的方法论。

1.2 企业数据资产

有了大数据的光环，有了从数据中挖掘商业价值的方法和工具之后，那些原本存放在服务器上平淡无奇的陈年旧数一夜之间身价倍增。按照世界经济论坛报告的看法，“大数据为新财富，价值堪比石油”。《大数据时代》一书的作者维克托则乐观地预测，数据列入企业资产负债表只是时间问题。

本质上，任何企业在生产活动中都会产生数据，数据都有分析的价值。我们来看看典型的运营商会产生哪些数据。

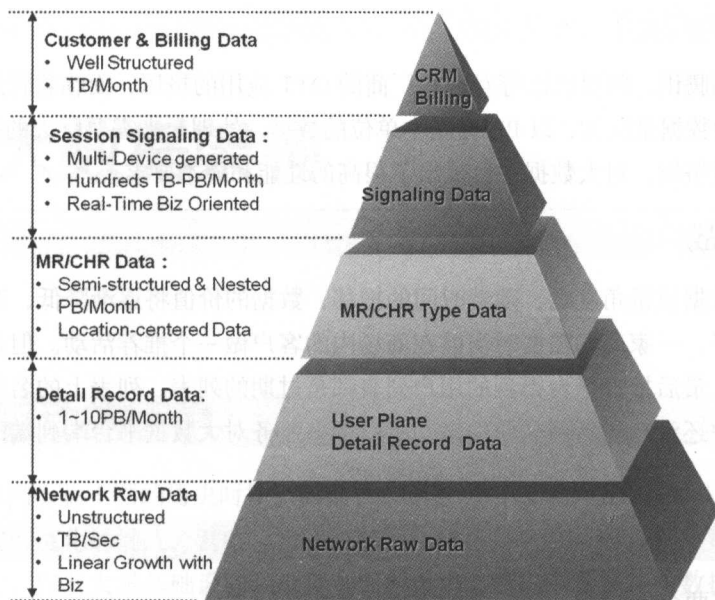
图 1.2 是典型的运营商产生的数据，从下到上分为如下几类。

- **Network Raw Data:** 电信网络里任何一个呼叫或者上网行为都会引起电信设备之间的数据进行交换，这个数据就是网络底数据。
- **User Plane Detail Record Data:** 从网络原始数据里面提取出来的用户行为数据，如打电话数据、上网行为等。
- **MR/CHR Type Data:** 无线测量数据、呼叫历史记录单据数据。用户的位置信息就是从 MR/CHR 数据里面通过算法得出的。
- **Signaling Data:** 信令数据。电信网络分控制面、数据面、用户面。在控制面上设备之间相互按照协议协商通信的数据叫信令数据。
- **CRM Billing:** 电信设备厂商登记的开户信息、账单信息。

上面的划分是从非常专业化的角度进行的，其实通俗一点可以简单归类为设备数据和用户数据两类。设备数据用来分析设备的正常与否，用于设备的维护、规划等。用户数据包括如下数据。

- **位置数据:** 无线是用一个个蜂窝来划分区域的，一个蜂窝叫一个小区，所以只要电话在线，就需要注册到一个个小区中去，知道了小区就知道了用户所在的位置。通过小区切换就能计算出用户移动的轨迹，这就是用户的位置数据。
- **上网数据:** 用户通过运营商的设备上网，所有的行为数据都可以被记录下来，如上了什么网站、网速是多少、上了多长时间。这些通过通信协议的包头就可以获取。如果继续分析内容，就可以获取更多的数据，就可以完全知道用户在干什么。

^① Data Warehouse, 数据仓库。



Note : In a typical network of ~30M subscribers and the data flow around 1TB/Sec.
MR/CHR: Measurement Report/Call History Data generated during calls

图 1.2^①

- 用户兴趣数据：通过用户的上网记录，就可以衍生出用户的兴趣爱好、常上什么网站、最近关注什么东西等。
- 通信数据和社交数据：例如，用户给谁打了电话、打了多长时间、给谁发了短信，这些信息都可以被记录下来。通过电话联系又可以衍生出用户的社交关系数据，如和谁是熟人、常联系谁等。
- 身份信息数据：用户到运营商开户，用户的所有个人信息就被运营商记录下来，包括姓名、年龄、身份证号码等。而且这些数据是由人工采集的，准确度远高于互联网用户自己注册的信息。
- 用户金融数据：如用户的电话、网络缴费记录，是否经常欠费等，可用于进行个人信用分析。

由此可以看出，运营商拥有从底层的设备数据到上层用户的行为数据，而且通常是全网的数据，因此可以说抱着数据的“金矿”，是其他厂商所无法媲美的。

1.3 大数据挑战

大数据发展到现在，有了一定的技术和商业积累，但是还有很多难题等待解答，最典型的的就是成本、实时性、安全等方面的挑战。

① 图片来源：<https://databricks.com/blog/2015/06/09/huawei-embraces-open-source-apache-spark.html>。

1.3.1 成本挑战

运营商普遍受到腾讯、阿里巴巴等互联网厂商的 OTT 应用的挤压，面临着管道转型、利润下降的风险。而运营商的数据量巨大，以 PB 为基本单位的数据，处理起来需要巨大的投入。外部商业环境和内部规模的双重挤压，对大数据平台提出了很高的性能和成本要求。

1.3.2 实时性挑战

如果从广义的数据质量角度看，随着时间的推移，数据的价值将逐渐降低，时间越久的数据，价值越低。举个例子，一家商场需要对当前在商场内的客户做一个推荐活动。但是端到端采集和处理数据的时间过长，最后推荐平台得到的用户列表都是过期的列表，列表上的名单可能已经不在商场内，而新到的用户还没有更新到名单中来，所以很多业务对大数据平台端到端的实时性提出了很高的要求。

1.3.3 安全挑战

安全挑战体现在两个方面：

一方面是在技术上，随着 HTTPS 的推广应用，数据在传输过程中采用管理加密的方式，运营商作为管道获取数据的难度变得越来越大。

另一方面是在法理上，用户的哪些数据是可以获取的、哪些是不允许读取的，始终存在侵犯用户隐私的法律风险。

1.4 小结

本章简述大数据的历史，探讨大数据的作用，作为全书的引子，让读者对大数据有一个初步的认识，带领读者进入丰富多彩的大数据世界。

后面的章节将重点讲述在构建运营商大数据系统时的业务和架构实战经验，以实际经验以飨读者。

第2章

运营商大数据架构

2.1 架构驱动的因素

运营商和互联网面临不同的历史时期，因而大数据在各自领域承担的使命是不一样的。

运营商面临被管道化的挑战，营收下滑，大数据项目承担企业战略转型、数据变现的使命。同时由于成本的压力，以及大量基础设施和设备利旧的诉求，所以运营商在大数据项目中，对性能、成本和集成度提出了很高的要求。

互联网企业近几年盈利颇丰，大数据往往是承担业务快速创新、未来探索的一种驱动因素，所以对架构的扩展性、灵活性等方面的追求优先级在成本之上。互联网企业每建一个数据中心通常就是几千台的规模，这在运营商看来是不可想象的。

背后的商业驱动因素不一样，所带来的架构挑战也不一样。

2.2 大数据平台架构

本书我们将以一个实际的大数据架构参与者、旁观者的角色讲述真正的实战经验，希望带给读者一些启发。前面讲到商业驱动因素不一样，所面临的场景不一样，选择的技术措施也会有所区别，但是其实存在即合理，实践出真知。

大数据平台架构如图 2.1 所示。可以看到，最上层是应用，大数据平台最后还是要解决实际的业务问题，在运营商领域分别解决 SQM（运维质量管理）、CSE（客户体验提升）、MSS（市场运维支撑）、DMP（数据管理平台）等问题。这部分内容会在第 3 章详细介绍。

第二层是各个组件/技术支撑，包括数据从产生获取、处理（实时、批处理）、分析（交互式查询、机器学习与数据挖掘）到最后的展现。这部分内容会在第 4~8 章介绍。

第三层，为了支持数据的存储处理，需要统一的资源管理及分配。这部分内容会在第 9 章介绍。

第四层，上层框架和处理都构建在存储的基础上，所以存储是基础中的基础。这部分内容会在第 10 章介绍。

第五层，大数据部署形态有云化部署、物理机部署等多种部署模式。这部分内容会在第 11 章介绍。

第 12 章介绍大数据技术开发文化。

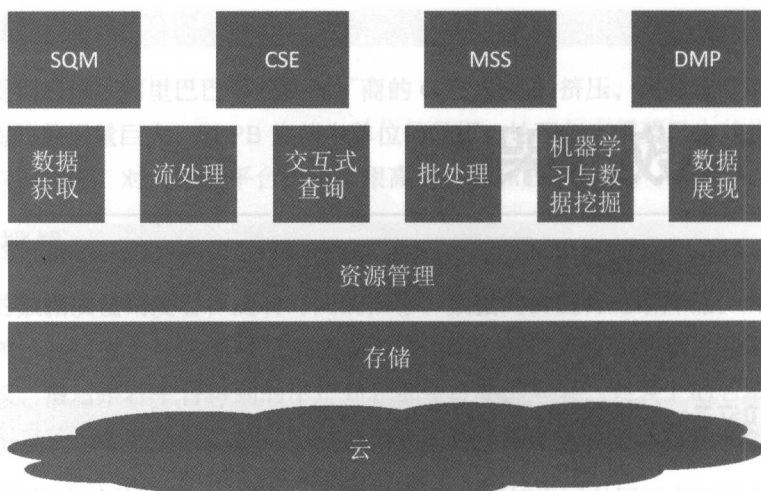


图 2.1

2.3 平台发展趋势

Hadoop 从 2006 年项目成立开始，已经风风雨雨走过了 10 年，从最开始的 HDFS 和 MapReduce 两个组件到现在完整的生态链。展望未来，随着技术和业务的发展，下面这些趋势应该是所有设计和实现大数据平台的人需要认真考虑的。

- Cloud First: 云优先。服务端利用云的部署和扩展能力，保证数据访问高并发、高可用、高可靠。
- Stream Default: 流优先。数据源端更多的是流数据，要求实时分析，进行秒级或分钟级计算。
- Pervasive Analytics: 普适分析。将分析能力推至数据源端、管道和服务端，低时延反馈结果。
- Self Service: 自服务。无须太多的人为干预和人力投入，使得数据合理放置，转换为适合分析的数据类型，方便 APP 开发等。

现在看着风光无限的组件或者平台，会不断地被后来者所替代。

2.4 小结

本章简要总结了本书的主要章节和内容。本书是围绕一个通用的大数据处理逻辑架构来展开的。在实际的生产环境中，该架构并不是一成不变的，会根据业务来灵活地部署和应用。当然，在一个完整的企业大数据系统里，本书介绍的内容完全不够，本书只介绍最基础的大数据平台，很多底层或者上层的内容可能没有覆盖到。另外，架构不是凭空出现的，由业务场景驱动的架构才是真正可用的架构。

第 3 章

运营商大数据业务

前面讲到，运营商由于处于网络管道的位置，因而掌握了非常多的数据，运营商开展大数据业务是顺理成章的事，而事实上大数据也承载着运营商向数字运营商转型的战略梦想。本章简要介绍运营商常见的大数据业务，让读者对大数据能做什么有一个基本的认识。

3.1 运营商常见的大数据业务

第 1 章介绍了运营商拥有的数据资产，运营商拥有从底层的设备和网络数据到上层的用户行为数据。有了这些数据，运营商大数据便可以衍生出众多业务，主要有 SQM（运维质量管理）、CSE（客户体验提升）、MSS（市场运维支撑）、DMP（数据管理平台）。

3.1.1 SQM（运维质量管理）

一个电话或者用户一个上网行为的成功发生，对于整个运营商背后的网络来说，经过了很多种类、很多台设备才能一起完成。传统的监控和告警设备只能单点地监控整个流程中的某一处是否出现问题，这样就和业务脱钩了。

以在线视频播放为例，这是用户常用的业务类型之一。传统设备只能监控单点的 KPI，但这和业务的实际体验其实是脱钩的，如核心网侧速度很高，但是用户体验仍可能很差，经常卡顿，所以运营商也很难判断出整个业务的质量好坏。通过大数据技术，不仅可以收集全网的各台设备的数据，而且可以收集设备原始数据，监控和计算出“播放成功率”、“缓存时延”、“停顿频次”、“停顿占比”、“中断率”等指标数据，就可以知道整个业务的质量如何。同时对各项指标不正常的具体原因，如中断率过高，可以通过收集的数据进一步钻取数据定位。

SQM 主要涉及的技术就是后分析，通过计算 KPI/KQI^①，得出各个业务的质量，支撑整个 SQM。

3.1.2 CSE（客户体验提升）

用户是互联网发展的基石，服务好用户是网络建设的目标。通过大数据技术，可以针对用户的个人体验进行 360°分析。用户体验提升一般包括如下几个方面：

^① KPI/KQI: Key Performance Indicators, 关键业绩指标; Key Quality Indicators, 关键质量指标。

(1) 通过对用户语音、短消息、流媒体等业务进行分析，定位到用户的哪个业务、哪类事件、哪个区域出现了问题，并呈现单个用户的活动轨迹，定位失败位置，以及对失败原因进行详细分析，为维护分析人员进行业务问题定位提供详尽的依据。

(2) 可以针对 VIP 用户，以全网为监控对象，对反映 VVIP 用户的 KPI 指标进行 1 分钟粒度的实时数据监控，帮助运营商实时了解 VVIP 用户的业务质量；以全网为监控对象，对反映用户群的 KPI 指标进行 5 分钟粒度的数据监控，并提供历史业务数据的失败分析，帮助运营商及时发现问题。

(3) 针对整个用户群，可以分别展示 CS、PS 等业务质量的总体情况，支持以不同维度对失败业务进行分析，并支持对异常原因进行分析，为保证用户/用户群的业务质量提供有力的数据支撑。

(4) 针对客户漫游专门进行监控。实时监控漫入/漫出客户数，对漫游的质量进行分析等。

CSE 是针对用户维度的业务和质量分析，涉及的技术是用户画像技术。

3.1.3 MSS（市场运维支撑）

前两类业务更多考虑的是运维支撑角度。除了运维支撑外，大数据还帮助运营商做好运营，如帮助运营商更好地掌握网络质量和提高客户满意度。常见的手段有离网分析。

简单介绍一下什么是离网分析。离网分析是通过按照客户群和客户来统计用户的业务数据，在定制报表服务器中对数据进行简易模型计算，给出用户的业务评分和质差事件评分，更好地帮助运营商及时预测用户离网倾向。

离网分析主要用到数据挖掘技术，通过建模，从数据中挖掘出用户离网倾向的主要原因，以及指导后续的改进。

3.1.4 DMP（数据管理平台）

运营商掌握了一部分数据。除了运营、运维的支撑外，数据变现也是其中重要的一部分，这通常是通过 DMP（数据管理平台）来实现的。

DMP 平台更多的是一种数据管理和变现的商业模式，技术上没有特殊的地方，由普通的 Hadoop、数据库来承担。

DMP（Data Management Platform）是把分散的第一、第三方数据进行整合，纳入统一的技术平台，并对这些数据进行标准化和细分，让用户可以把这些细分结果推向现有的互动营销环境中。

DMP 的核心元素如下。

- 数据整合及标准化能力：采用统一化的方式，将各方数据吸纳整合。
- 数据细化管理能力：创建出独一无二、有意义的客户细分，进行有效的营销活动。
- 功能健全的数据标签：提供数据标签灵活性，便于营销活动使用。
- 自助式的用户界面：用户可以基于 Web 界面或其他集成方案来直接使用数据工具、相关功能和几张报表等。

- 相关渠道环境的连接：与相关渠道集成，包含网站端、展示广告、电子邮件及搜索和视频，让营销者能找到、定位和提供细分群体相关高度的营销信息。

DMP 最重要的技术是标签管理和存储，标签的维度非常大，需要一个好的引擎支持标签的存储、更新及快速对外提供服务。

3.2 小结

本章高度概括了运营商的一些典型业务，帮助读者了解大数据能帮助运营商做什么，但是运营时的业务远比这个复杂。运营商掌握了大量的数据，目前的业务涵盖运维、运营、数据变现等各个方面，大数据的各种技术都会用到，后面的章节会介绍从数据获取到数据应用所涉及的技术。但总的来说，相比互联网企业，运营商使用大数据还是偏传统的应用，局限于运营商本身。以大数据为基础，相信未来运营商在商业模式上发挥的空间还是很大的。

第一章 第

題 第 1.4

第二部分 大数据技术

第 4 章

数据获取

大数据技术的核心是从数据中获取价值，而第一步就是要弄清楚有什么数据、怎样获取。在企业生产过程中，数据无所不在，但是如果不能正确获取，或者没有能力获取，就浪费了宝贵的数据资源。本章主要介绍数据获取的技术。

4.1 数据分类

数据的分类方法有很多种，按数据形态可以分为结构化数据和非结构化数据两种。结构化数据如传统的 Data Warehouse 数据；非结构化数据有文本数据、图像数据、自然语言数据等。

结构化数据和非结构化数据的区别从字面上就很容易理解：结构化数据，结构固定，每个字段有固定的语义和长度，计算机程序可以直接处理；而非结构化数据，计算机程序无法直接处理，需要先对数据进行格式转换或信息提取。

按数据的来源和特点，电信的数据又可以分为网络原始数据、用户面详单信令、信令数据等。运营商的数据分类在 1.2 节介绍过，可以回顾一下。

运营商数据是一个数据“金矿”，包括用户数据和设备数据，但是运营商的数据又有如下特点：

- 数据种类复杂，结构化、半结构化、非结构化数据都有。运营商的设备由于传统设计的原因，很多都是根据协议来实现的，所以数据的结构化程度比较高，结构化数据易于分析，这点相比其他行业有天然的优势。
- 数据实时性要求高，如信令数据都是实时消息，如果不及时获取就会丢失。
- 数据来源广泛，各个设备数据产生的速度及传送速度都不一样，因而数据关联是一大难题。

让运营商数据产生价值的第一步是数据获取，下面介绍数据获取和数据分发的相关技术。

4.2 数据获取组件

数据的来源不同，数据获取涉及的技术也不同。电信的很多数据产生于网络设备，你会看到电

信特有的探针^①技术，以及为获取网页数据常用的爬虫、采集日志数据的组件 Flume；数据获取之后，为了方便分发到后面的系统处理，本章最后介绍常用的 Kafka^②消息中间件。

4.3 探针

4.3.1 探针原理

打电话，手机上网，背后承载的都是电信的路由器、交换机等设备的数据交换。从电信的路由器、交换机上把数据采集上来的专用设备是探针。根据探针放置的位置不同，可分为内置探针和外置探针两种。

- 内置探针：探针设备和电信已有设备部署在同一个机框内，直接获取数据。
- 外置探针：在现网中，大部分网络设备早已经部署完毕，无法移动原有网络，这时就需要外置探针。

外置探针主要由以下几个设备组成，如图 4.1 所示。

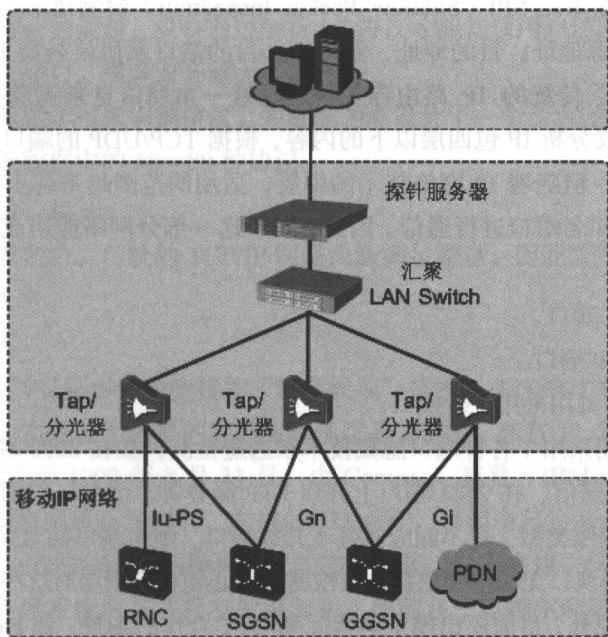


图 4.1

- Tap/分光器：对承载在铜缆、光纤上传输的数据进行复制，并且不影响原有两个网元^③间的数据传输。

① 探针：Probe 获取电信设备数据的专用设备，根据其与电信设备的位置关系，分内置探针和外置探针。

② Kafka：高吞吐量消息中间件（<https://kafka.apache.org/>）。

③ 网元：指两个电信网络设备实体。

- 汇聚 LAN Switch：汇聚多个 Tap/分光器复制的数据，上报给探针服务器。
- 探针服务器：对接收到的数据进行解析、关联等处理，生成 xDR^①，并将 xDR 上报给分析系统，作为其数据分析的基础。

探针通过分光器获取到数据网络中各个接口的数据，然后发送到探针服务器进行解析、关联等处理。经过探针服务器解析、关联的数据，最后送到统一分析系统中进行进一步的分析。

4.3.2 探针的关键能力

1. 大容量

探针设备需要和电信已有的设备部署在一起。一般来说，原有设备的机房空间有限，所以探针设备的高容量、高集成度是非常关键的能力。

探针负责截取网络数据并解析出来，其中最重要的是转发能力，对网络的要求很高。高性能网络是大容量的保证，后面将介绍探针用到的 IB^②技术。

2. 协议智能识别

传统的协议识别方法采用 SPI (Shallow Packet Inspection) 检测技术。SPI 对 IP 包头中的“5 Tuples”，即“五元组（源地址、目的地址、源端口、目的端口及协议类型）”信息进行分析，来确定当前流量的基本信息。传统的 IP 路由器正是通过这一系列信息来实现一定程度的流量识别和 QoS 保障的，但 SPI 仅仅分析 IP 包四层以下的内容，根据 TCP/UDP 的端口来识别应用。这种端口检测技术检测效率很高，但随着 IP 网络技术的发展，适用的范围越来越小，目前仍有一些传统网络应用协议使用固定的知名端口进行通信。因此，对于这一部分网络应用流量，可以采用端口检测技术进行识别。例如：

- DNS 协议采用 53 端口。
- BGP 协议采用 179 端口。
- MSRPC 远程过程调用采用 135 端口。

许多传统和新兴应用采用了各种端口隐藏技术来逃避检测，如在 8000 端口上进行 HTTP 通信、在 80 端口上进行 Skype 通信、在 2121 端口上开启 FTP 服务等。因此，仅通过第四层端口信息已经不能真正判断流量中的应用类型，更不能应对基于开放端口、随机端口甚至采用加密方式进行传输的应用类型。要识别这些协议，无法单纯依赖端口检测，而必须在应用层对这些协议的特征进行识别。

除了逃避检测的情况外，目前还出现了运营商和 OTT 合作的场景，如 Facebook 包月套餐，在这种情况下，运营商可以基于 OTT 厂商提供的 IP、端口等配置信息进行计费。但是这种方式有很大的限制，如系统配置的 IP 和端口数量有限、OTT 厂商经常改变或者增加服务器造成频繁修改配置等。

协议智能识别技术能够深度分析数据包所携带的 L3~L7/L7+ 的消息内容、连接的状态/交互信息

① xDR：TDR (Transaction Detail Record, 事务详细单据)、CDR (Call Detail Record, 呼叫详细单据) 的合称。

② IB：InfiniBand。

(如连接协商的内容和结果状态、交互消息的顺序等)等信息,从而识别出详细的应用程序信息(如协议和应用的名称等)。

3. 安全的影响

探针的核心能力是获取通信的数据,但随着越来越多的网站使用 HTTPS/QUIC^①加密 L7 协议,传统的探针能力就会受到极大的限制,因而无法解析 L7 协议的内容,如图 4.2 所示。

Network	解析					
L7+ Service						
L7 Application	Video Codec			NULL	解析失效	解析失效
L7 Protocol	Method	URL	Host	Body		
	Head					
	非加密(HTTP)				加密(HTTP)	加密(QUIC)
				SSL		
L4	TCP				UDP	
L3	IP					

图 4.2

比如,想分析 YouTube 的流量,只有通过解析 L7 协议才能知道用户访问的是 YouTube,所以加密之后会影响探针的解析能力,很多业务就无法进行。

现在业界尝试使用深度学习来识别协议,如奇虎 360 设计了一个 5~7 层的深度神经网络,能够自动学习特征并识别每天数据中的 50~80 种协议。

4. IB (InfiniBand) 技术

为了达到高效的转发能力,传统的 TCP/IP 网络无法满足需求,因此需要更高速度、更大带宽、更高效率的 InfiniBand 网络。

1) 什么是 IB 技术

InfiniBand 架构是一种支持多并发链接的“转换线缆”技术。在这种技术中,仅有一个链接的时候运行速度是 500MB/s,在有 4 个链接的时候运行速度是 2GB/s,在有 12 个链接的时候运行速度可以达到 6GB/s。IBTA 成立于 1999 年 8 月 31 日,由 Compaq、惠普、IBM、戴尔、英特尔、微软和 Sun 七家公司牵头,共同研究高速发展的、先进的 I/O 标准。最初命名为 System I/O,1999 年 10 月正式更名为 InfiniBand。InfiniBand 是一种长缆线的连接方式,具有高速、低延迟的传输特性。

InfiniBand 用于服务器系统内部并没有发展起来,原因在于英特尔和微软在 2002 年就退出了 IBTA。在此之前,英特尔早已另行倡议 Arapahoe,也称为 3GIO (3rd Generation I/O, 第三代 I/O),即今日鼎鼎大名的 PCI-Express (PCI-E)。InfiniBand、3GIO 经过一年的并行,英特尔最终选择了 PCI-E。因此,现在应用 InfiniBand,主要用于服务器集群、系统之间的互联。

① QUIC (Quick UDP Internet Connections, 发音“quick”)是一种基于 UDP 的多路传输协议,它的主要目标是实现零往返时间的连接开销。

2) IB 速度快的原因

随着 CPU 性能的飞速发展, I/O 系统的性能成为制约服务器性能的瓶颈, 于是人们开始重新审视使用了十几年的 PCI 总线架构。虽然 PCI 总线架构把数据的传输从 8 位/16 位一举提升到 32 位, 甚至当前的 64 位, 但是它的一些先天劣势限制了其继续发展的势头。PCI 总线有如下缺陷:

(1) 由于采用了基于总线的共享传输模式, 所以在 PCI 总线上不可能同时传送两组以上的数据, 当一个 PCI 设备占用总线时, 其他设备只能等待。

(2) 随着总线频率从 33MHz 提高到 66MHz, 甚至 133MHz (PCI-X), 信号线之间的相互干扰变得越来越严重, 在一块主板上布设多条总线的难度也就越来越大。

(3) 由于 PCI 设备采用了内存映射 I/O 地址的方式建立与内存的联系, 热添加 PCI 设备变成了一件非常困难的工作。目前的做法是在内存中为每个 PCI 设备划出一块 50~100MB 的区域, 这段空间用户是不能使用的。因此, 如果一块主板上支持的热插拔 PCI 接口越多, 用户损失的内存就越多。

(4) PCI 总线上虽然有 Buffer 作为数据的缓冲区, 但是它不具备纠错的功能。如果在传输过程中发生了数据丢失或损坏的情况, 则控制器只能触发一个 NMI 中断通知操作系统在 PCI 总线上发生了错误。

3) IB 介绍^①

① InfiniBand 架构

InfiniBand 架构如图 4.3 所示。

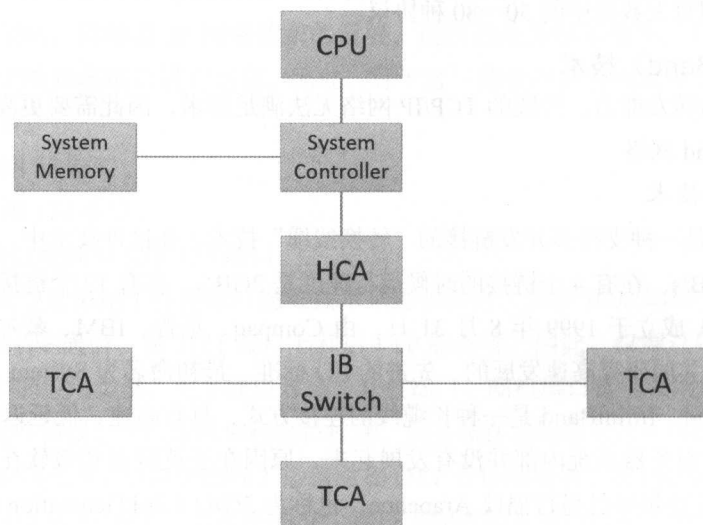


图 4.3

InfiniBand 采用双队列程序提取技术, 使应用程序直接将数据从适配器送入应用内存 (远程直接存储器存取, RDMA), 反之亦然。在 TCP/IP 协议中, 来自网卡的数据先复制到核心内存, 然后再

^① 参考 Mellanox 技术白皮书 *Introduction to InfiniBand*。

复制到应用存储空间，或从应用存储空间将数据复制到核心内存，再经由网卡发送到 Internet。这种 I/O 操作方式始终需要经过核心内存的转换，不仅增加了数据流传输路径的长度，而且大大降低了 I/O 的访问速度，增加了 CPU 的负担。而 SDP 则是将来自网卡的数据直接复制到用户的应用存储空间，从而避免了核心内存的参与。这种方式被称为零拷贝，它可以在进行大量数据处理时，达到该协议所能达到的最大吞吐量。

InfiniBand 的协议采用分层结构，各个层次之间相互独立，下层为上层提供服务。其中，物理层定义了在线路上如何将比特信号组成符号，然后再组成帧、数据符号及包之间的数据填充等，详细说明了构建有效包的信令协议等；链路层定义了数据包的格式及数据包操作的协议，如流控、路由选择、编码、解码等；网络层通过在数据包上添加一个 40 字节的全局的路由报头(Global Route Header, GRH)来进行路由的选择，对数据进行转发，在转发过程中，路由器仅仅进行可变的 CRC 校验，这样就保证了端到端数据传输的完整性；传输层再将数据包传送到某个指定的队列偶(Queue Pair, QP)，并指示 QP 如何处理该数据包，以及当信息的数据净核部分大于通道的最大传输单元(MTU)时，对数据进行分段和重组，如图 4.4 所示。

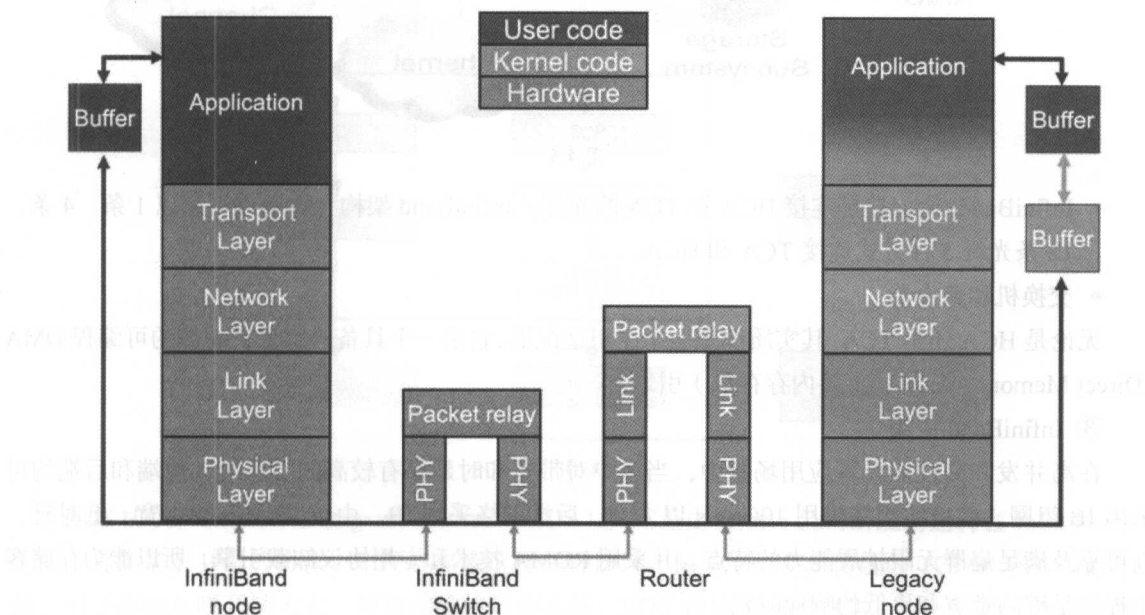


图 4.4

② InfiniBand 基本组件

InfiniBand 的网络拓扑结构如图 4.5 所示，其组成单元主要分为 4 类。

- HCA (Host Channel Adapter)：它是连接内存控制器和 TCA 的桥梁。
- TCA (Target Channel Adapter)：它将 I/O 设备（如网卡、SCSI 控制器）的数字信号打包发送给 HCA。

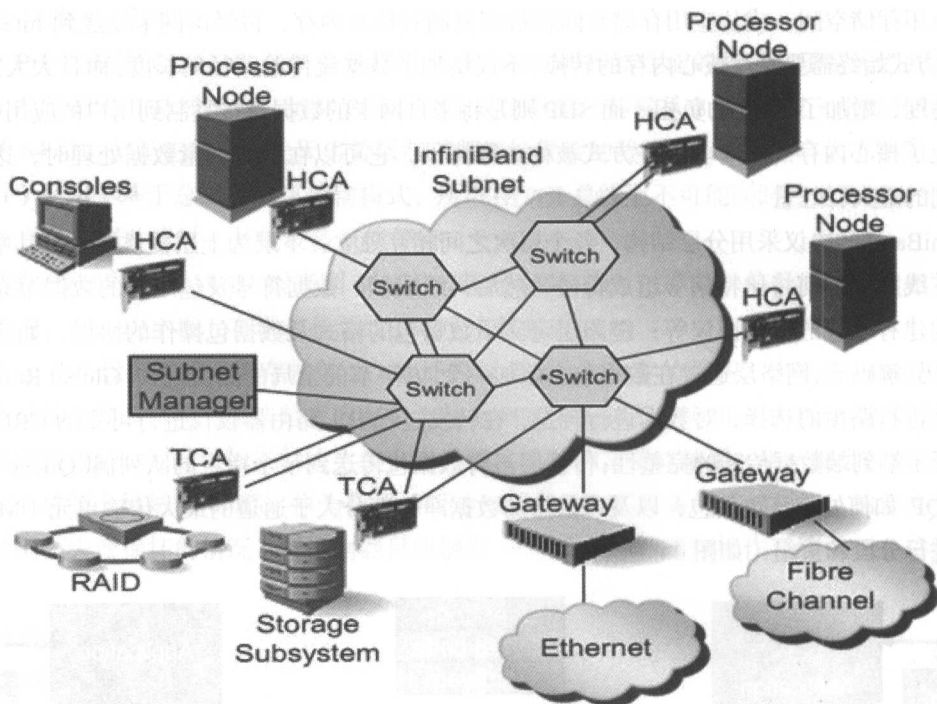


图 4.5

- InfiniBandlink：它是连接 HCA 和 TCA 的光纤。InfiniBand 架构允许硬件厂家以 1 条、4 条、12 条光纤 3 种方式连接 TCA 和 HCA。
- 交换机和路由器。

无论是 HCA 还是 TCA,其实质都是一个主机适配器,它是一个具备一定保护功能的可编程 DMA (Direct Memory Access, 直接内存存取) 引擎。

③ InfiniBand 应用

在高并发和高性能计算应用场景中,当客户对带宽和时延都有较高的要求时,前端和后端均可采用 IB 组网,或前端网络采用 10Gbit/s 以太网,后端网络采用 IB。由于 IB 具有高带宽、低时延、高可靠及满足集群无限扩展能力的特点,并采用 RDMA 技术和专用协议卸载引擎,所以能为存储客户提供足够的带宽和更低的响应时延。

IB 目前可以实现及未来规划的更高带宽工作模式如下(以 4X 模式为例)。

- SRD (Single Data Rate): 单倍数据率,即 8Gbit/s。
- DDR (Double Data Rate): 双倍数据率,即 16Gbit/s。
- QDR (Quad Data Rate): 4 倍数据率,即 32Gbit/s。
- FDR (Fourteen Data Rate): 14 倍数据率,即 56Gbit/s。
- EDR (Enhanced Data Rate): 100Gbit/s。

- HDR (High Data Rate): 200Gbit/s。
- NDR (Next Data Rate): 1000Gbit/s+。

4) IB 常见的运行协议^①

IPoIB 协议: Internet Protocol over InfiniBand, 简称 IPoIB。传统的 TCP/IP 栈的影响实在太大了, 几乎所有的网络应用都是基于此开发的, IPoIB 实际是 InfiniBand 为了兼容以太网不得不做的一种折中, 毕竟谁也不愿意使用不兼容大规模已有设备的产品。IPoIB 基于 TCP/IP 协议, 对于用户应用程序是透明的, 并且可以提供更大的带宽, 也就是原先使用 TCP/IP 协议栈的应用不需要任何修改就能使用 IPoIB 协议。例如, 如果使用 InfiniBand 做 RAC 的私网, 默认使用的就是 IPoIB 协议。图 4.6 左侧是传统以太网 TCP/IP 协议栈的拓扑结构, 右侧是 InfiniBand 使用 IPoIB 协议的拓扑结构。

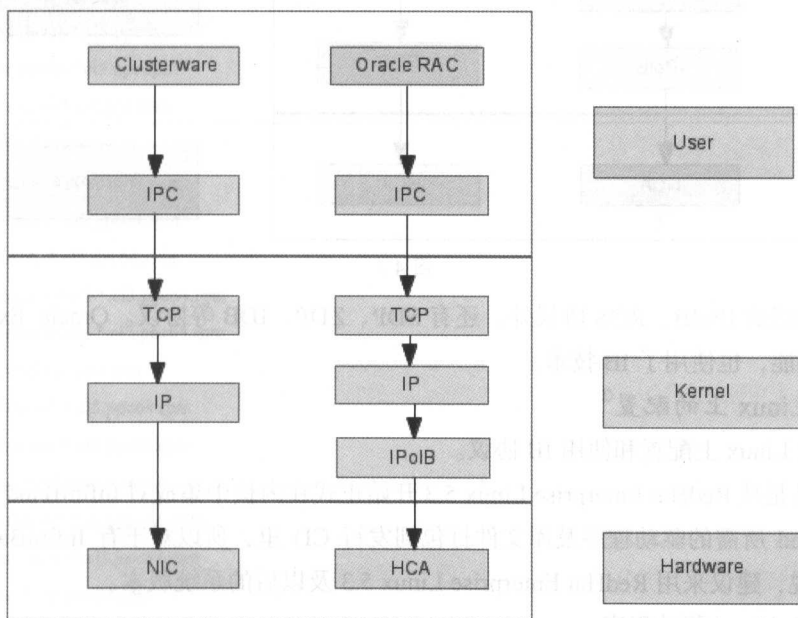


图 4.6

RDS 协议: Reliable Datagram Sockets (RDS) 实际是由 Oracle 公司研发的运行在 InfiniBand 之上的、直接基于 IPC 的协议。之所以出现这样一种协议, 根本原因在于传统的 TCP/IP 栈本身过于低效, 对于高速互联开销太大, 导致传输的效率太低。RDS 相比 IPoIB, CPU 的消耗量减少了 50%; 相比传统的 UDP 协议, 网络延迟减少了一半。图 4.7 左侧是使用 IPoIB 协议的 InfiniBand 设备的拓扑结构, 右侧是使用 RDS 协议的 InfiniBand 设备的拓扑结构。在默认情况下, RDS 协议不会被使用, 需要进行额外的 relink (重新链接)。另外, 即使 relink RDS 库以后, RAC 节点间的 CSS 通信也无法使用 RDS 协议, 节点间的心跳维持及监控都采用 IPoIB 协议。

^① 参考 <http://storage.chinabyte.com/155/12533155.shtml>。

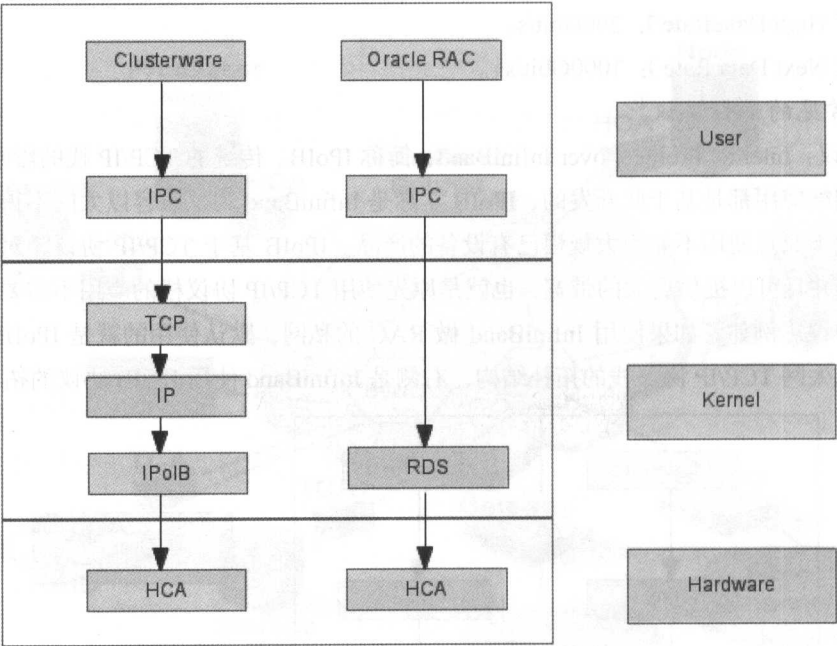


图 4.7

除了上面介绍的 IPoIB、RDS 协议外，还有 SDP、ZDP、IDB 等协议。Oracle Exadata 一体机为了达到较高的性能，也使用了 IB 技术。

5) IB 在 Linux 上的配置^①

下面介绍在 Linux 上配置和使用 IB 协议。

RedHat 产品是从 RedHat Enterprise Linux 5.3 开始正式在内核中集成对 InfiniBand 网卡的支持的，并且将 InfiniBand 所需的驱动程序及库文件打包到发行 CD 里，所以对于有 InfiniBand 应用需求的 RedHat 用户来说，建议采用 RedHat Enterprise Linux 5.3 及以后的系统版本。

① 安装 InfiniBand 驱动程序

在安装 InfiniBand 驱动程序之前，首先确认 InfiniBand 网卡已经被正确地连接或分配到主机，然后从 RedHat Enterprise Linux 5.3 的发行 CD 中获得 Table1 中给出的 RPM 文件，并根据上层应用程序的需要，选择安装相应的 32 位或 64 位软件包，如表 4.1 所示。

表 4.1

openib-*.el5.noarch.rpm	
32bit	libibcm-*.el5.ppc.rpm
	libibcm-devel-*.el5.ppc.rpm
	libibcm-static-*.el5.ppc.rpm
	libibcommon-*.el5.ppc.rpm

① 参考 <http://www.ibm.com/developerworks/cn/linux/l-cn-infiniband/>

续表

openib-*.el5.noarch.rpm

32bit	libibcommon-devel-*.el5.ppc.rpm
	libibcommon-static-*.el5.ppc.rpm
	libibmad-*.el5.ppc.rpm
	libibmad-devel-*.el5.ppc.rpm
	libibmad-static-*.el5.ppc.rpm
	mlibibumad-*.el5.ppc.rpm
	libibumad-devel-*.el5.ppc.rpm
	libibumad-static-*.el5.ppc.rpm
	libibverbs-*.el5.ppc.rpm
	libibverbs-devel-*.el5.ppc.rpm
64bit	libibverbs-static-*.el5.ppc.rpm
	libibverbs-utils-*.el5.ppc.rpm
	libibcm-*.el5.ppc64.rpm
	libibcm-devel-*.el5.ppc64.rpm
	libibcm-static-*.el5.ppc64.rpm
	libibcommon-*.el5.ppc64.rpm
	libibcommon-devel-*.el5.ppc64.rpm
	libibcommon-static-*.el5.ppc64.rpm
	libibmad-*.el5.ppc64.rpm
	libibmad-devel-*.el5.ppc64.rpm
	libibmad-static-*.el5.ppc64.rpm
	libibumad-*.el5.ppc64.rpm
	libibumad-devel-*.el5.ppc64.rpm
	libibumad-static-*.el5.ppc64.rpm
	libibverbs-*.el5.ppc64.rpm
	libibverbs-devel-*.el5.ppc64.rpm
	libibverbs-static-*.el5.ppc64.rpm
	libibverbs-utils 64bit rpm is not available in RedHatEL5.3

另外，对于不同类型的 InfiniBand 网卡，还需要安装一些特殊的驱动程序。例如，对于 Galaxy1/Galaxy2 类型的 InfiniBand 网卡，就需要安装 ehca 相关的驱动。这里给出三种常见的 InfiniBand 网卡及其所需驱动的对应关系，如表 4.2 所示。

表 4.2

libehca (for Galaxy1/Galaxy2 support)	32bit	libehca-*.el5.ppc.rpm
		libehca-static-*.el5.ppc.rpm
	64bit	libehca-*.el5.ppc64.rpm
		libehca-static-*.el5.ppc64.rpm

:00

③ 配置 IPoIB 网络接口

上网接口的大

接口配置文件示例:

时可以使用

, 则可以参

4.4 网页采集

大量的数据散落在互联网中，要分析互联网上的数据，需要先把数据从网络中获取下来，这就需要网络爬虫技术。

4.4.1 网络爬虫^①

1. 基本原理

网络爬虫是搜索引擎抓取系统的重要组成部分。爬虫的主要目的是将互联网上的网页下载到本地，形成一个或联网内容的镜像备份。下面主要对爬虫及抓取系统的原理进行基本介绍。

一个通用的网络爬虫的框架如图 4.8 所示。

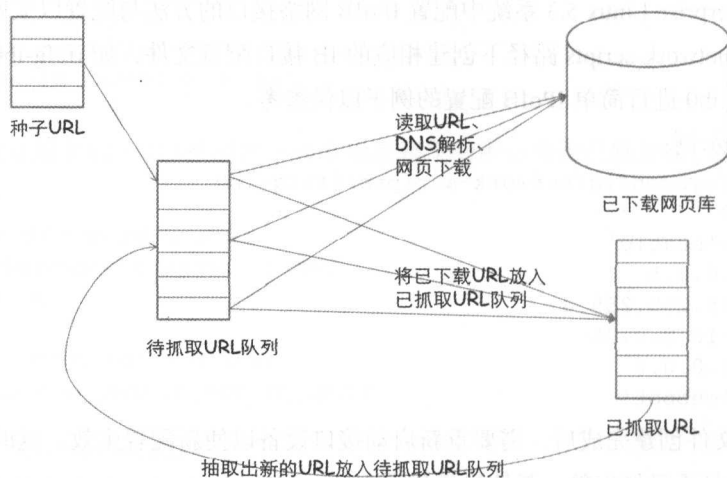


图 4.8

网络爬虫的基本工作流程如下：

- 首先选取一部分种子 URL。
- 将这些 URL 放入待抓取 URL 队列。
- 从待抓取 URL 队列中取出待抓取的 URL，解析 DNS，得到主机的 IP，并将 URL 对应的网页下载下来，存储到已下载网页库中。此外，将这些 URL 放入已抓取 URL 队列。
- 分析已抓取到的网页内容中的其他 URL，并且将 URL 放入待抓取 URL 队列，从而进入下一个循环。

从爬虫的角度对互联网进行划分，可以将互联网的所有页面分为 5 个部分，如图 4.9 所示。

- 已下载未过期网页。

^① 参考《网络爬虫基本原理》(<http://www.cnblogs.com/wawlian/tag/%E6%90%9C%E7%B4%A2%E5%BC%95%E6%93%8E/>)。

- 已下载已过期网页：抓取到的网页实际上是互联网内容的一个镜像与备份。互联网是动态变化的，一部分互联网上的内容已经发生变化，这时，这部分抓取到的网页就已经过期了。
- 待下载网页：也就是待抓取 URL 队列中的那些页面。
- 可知网页：还没有抓取下来，也没有在待抓取 URL 队列中，但是可以通过对已抓取页面或者待抓取 URL 对应页面进行分析获取到 URL，这些网页被称为可知网页。
- 还有一部分网页爬虫是无法直接抓取下载的，这些网页被称为不可知网页。

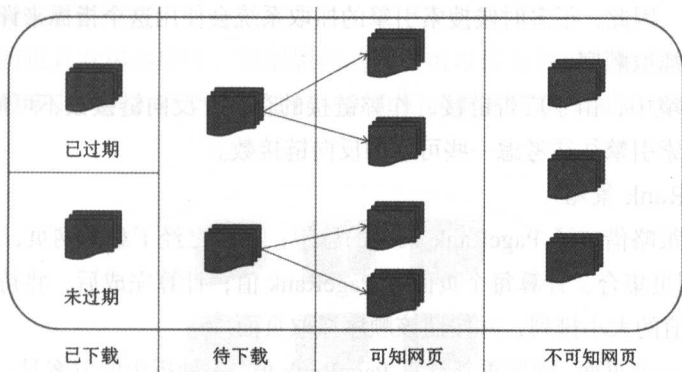


图 4.9

2. 抓取策略

在爬虫系统中，待抓取 URL 队列是很重要的一部分。待抓取 URL 队列中的 URL 以什么样的顺序排列也是一个很重要的问题，因为其决定了先抓取哪个页面、后抓取哪个页面。而决定这些 URL 排列顺序的方法叫作抓取策略。下面重点介绍几种常见的抓取策略。

1) 深度优先遍历策略

深度优先遍历策略是指网络爬虫会从起始页开始，一个链接一个链接地跟踪下去，处理完这条线路之后再转入下一个起始页，继续跟踪链接。以图 4.10 为例，遍历的路径为：A→F→GE→H→IBCD。

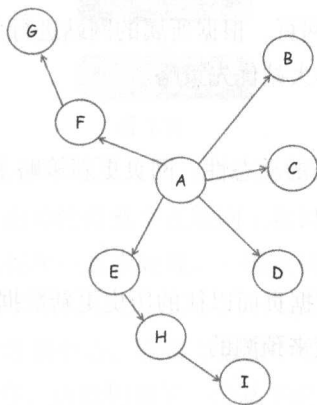


图 4.10

2) 宽度优先遍历策略

宽度优先遍历策略的基本思路是：将新下载网页中发现的链接直接插入待抓取 URL 队列的末尾。也就是说，网络爬虫会先抓取起始网页中链接的所有网页，然后再选择其中的一个链接网页，继续抓取此网页中链接的所有网页。仍以图 4.10 为例，遍历的路径为： $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow FGHI$ 。

3) 反向链接数策略

反向链接数是指一个网页被其他网页链接指向的数量。反向链接数表示的是一个网页的内容受到其他人推荐的程度。因此，很多时候搜索引擎的抓取系统会使用这个指标来评价网页的重要程度，从而决定不同网页的抓取顺序。

在真实的网络环境中，由于广告链接、作弊链接的存在，反向链接数不可能完全等同于网页的重要程度。因此，搜索引擎往往考虑一些可靠的反向链接数。

4) PartialPageRank 策略

PartialPageRank 策略借鉴了 PageRank 策略的思想：对于已经下载的网页，连同待抓取 URL 队列中的 URL，形成网页集合，计算每个页面的 PageRank 值；计算完成后，将待抓取 URL 队列中的 URL 按照 PageRank 值的大小排列，并按照该顺序抓取页面。

如果每次只抓取一个页面，则要重新计算 PageRank 值。一种折中的方案是：每抓取 K 个页面后，重新计算一次 PageRank 值。但是这种情况还会产生一个问题：对于已经下载下来的页面中分析出的链接，也就是未知网页部分，暂时是没有 PageRank 值的。为了解决这个问题，会赋予这些页面一个临时的 PageRank 值：将这个网页所有入链传递进来的 PageRank 值进行汇总，这样就形成了该未知页面的 PageRank 值，从而参与排序。

5) OPIC 策略

该策略实际上也是对页面进行重要性打分。在策略开始之前，给所有页面一个相同的初始现金 (cash)。当下载了某个页面 P 之后，将 P 的现金分摊给所有从 P 中分析出的链接，并且将 P 的现金清空。对于待抓取 URL 队列中的所有页面，按照现金数进行排序。

6) 大站优先策略

对于待抓取 URL 队列中的所有网页，根据所属的网站进行分类；对于待下载页面数多的网站，则优先下载。这种策略也因此被叫作大站优先策略。

3. 更新策略

互联网是实时变化的，具有很强的动态性。网页更新策略主要用来决定何时更新之前已经下载的页面。常见的更新策略有以下三种。

1) 历史参考策略

顾名思义，历史参考策略是指根据页面以往的历史更新数据，预测该页面未来何时会发生变化。一般来说，是通过泊松过程进行建模来预测的。

2) 用户体验策略

尽管搜索引擎针对某个查询条件能够返回数量巨大的结果，但是用户往往只关注前几页结果。

因此，抓取系统可以优先更新那些在查询结果中排名靠前的网页，然后再更新排名靠后的网页。这种更新策略也需要用到历史信息。用户体验策略保留网页的多个历史版本，并且根据过去每次的内容变化对搜索质量的影响得出一个平均值，将该值作为决定何时重新抓取的依据。

3) 聚类抽样策略

前面提到的两种更新策略都有一个前提：需要网页的历史信息。这样就会存在两个问题：第一，系统如果为每个网页保存多个历史版本信息，则无疑增加了系统负担；第二，如果新的网页完全没有历史信息，则无法确定更新策略。

这种策略认为，网页具有很多属性，类似属性的网页可以认为其更新频率也是类似的。要计算某个类别网页的更新频率，只需对这类网页抽样，以它们的更新周期作为整个类别的更新周期。基本思路如图 4.11 所示。

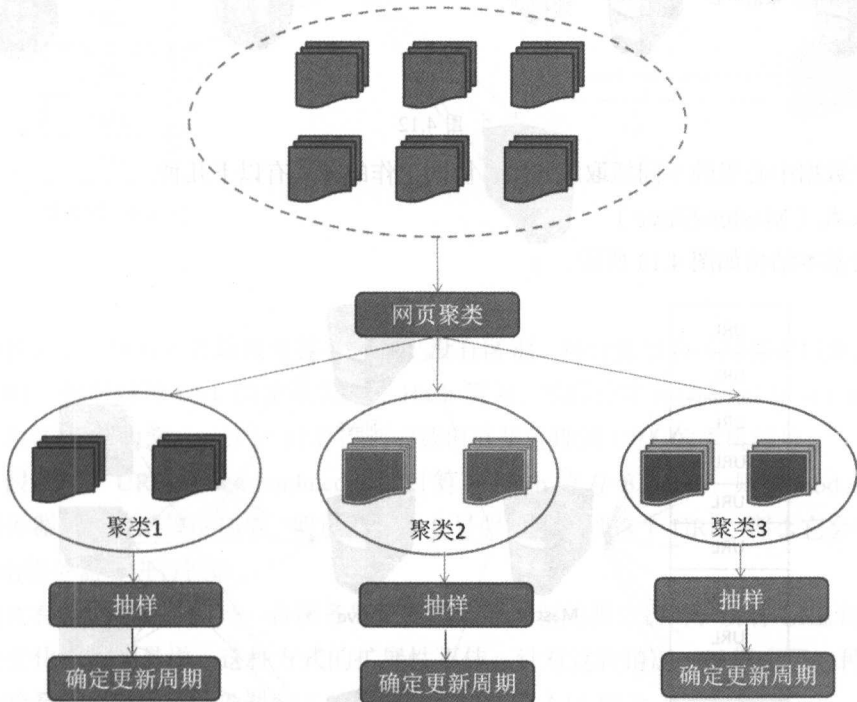


图 4.11

4. 系统架构

一般来说，分布式抓取系统需要面对的是整个互联网上数以亿计的网页，单个抓取程序不可能完成这样的任务，往往需要多个抓取程序一起来处理。一般来说，抓取系统往往是一个分布式的三层结构，如图 4.12 所示。

最底层是分布在不同地理位置的数据中心，在每个数据中心里有若干台抓取服务器，而每台抓取服务器上可能部署了若干套爬虫程序，这就构成了一个基本的分布式抓取系统。

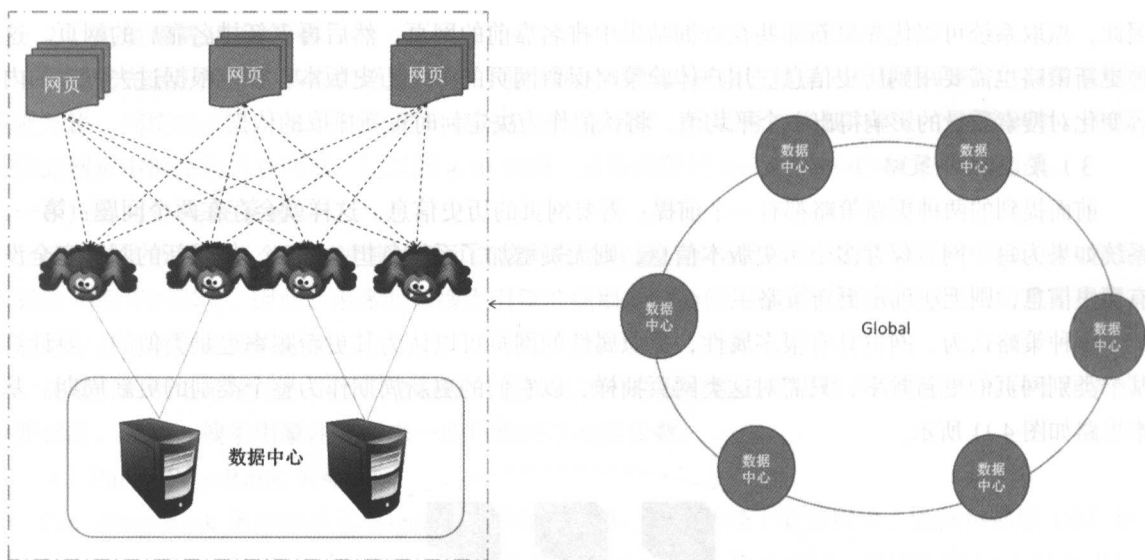


图 4.12

对于一个数据中心里的不同抓取服务器，协同工作的方式有以下几种。

1) 主从式 (Master-Slave)

主从式的基本结构如图 4.13 所示。

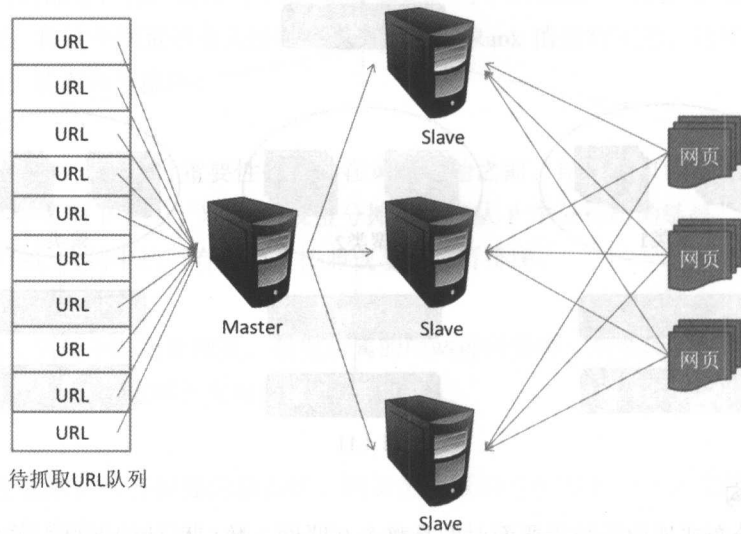


图 4.13

对于主从式而言，有一台专门的 Master 服务器来维护待抓取 URL 队列，它负责每次将 URL 分发到不同的 Slave 服务器，而 Slave 服务器则负责实际的网页下载工作。Master 服务器除了维护待抓取 URL 队列及分发 URL 外，还要负责调解各 Slave 服务器的负载情况，以免某些 Slave 服务器过于清闲或者劳累。

在这种模式下，Master 往往容易成为系统瓶颈。

2) 对等式 (Peer to Peer)

对等式的基本结构如图 4.14 所示。

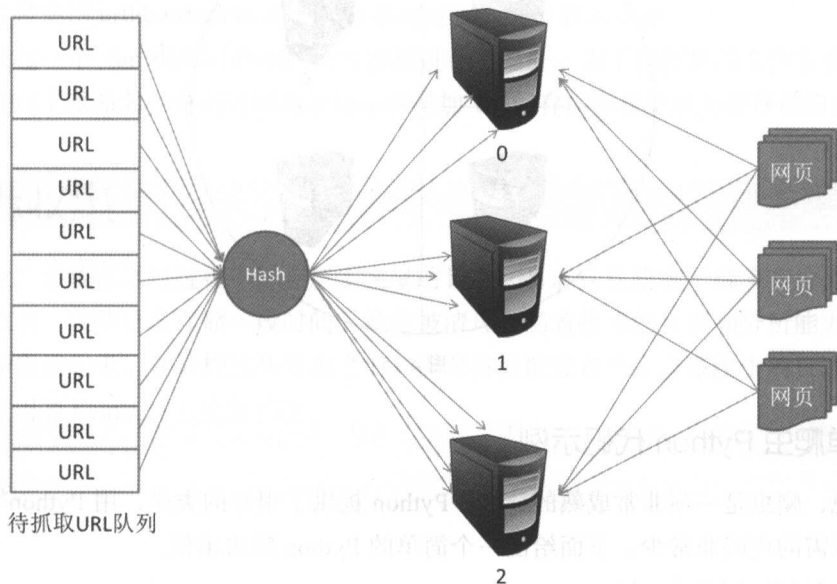


图 4.14

在这种模式下，所有的抓取服务器在分工上没有区别。每台抓取服务器都可以从待抓取 URL 队列中获取 URL，然后对该 URL 的主域名计算 Hash 值 H ，然后计算 H 值 $\bmod m$ （其中， m 是服务器的数量，以图 4.14 为例， $m=3$ ），计算得到的数值就是处理该 URL 的主机编号。

举例：假设对于 URL “www.baidu.com”，计算其 Hash 值 $H=8$ ， $m=3$ ，则 $H \bmod m=2$ ，因此由编号为 2 的服务器进行该链接的抓取。假设这时由 0 号服务器拿到这个 URL，那么它会将该 URL 转给服务器 2，由服务器 2 进行抓取。

这种模式有一个问题，即当一台服务器死机或者添加新的服务器时，所有 URL 的哈希求余的结果都将发生变化。也就是说，这种方式的扩展性不佳。针对这种情况，又提出了一种改进方案，即使用一致性哈希算法来确定服务器分工。其基本结构如图 4.15 所示。

一致性哈希算法对 URL 的主域名进行哈希运算，映射为范围在 $0 \sim 232$ 之间的某个数；然后将这个范围平均分配给 m 台服务器，根据 URL 主域名哈希运算的值所处的范围判断由哪台服务器来进行抓取。

如果某台服务器出现问题，那么原本由该服务器负责的网页则按照顺时针顺延，由下一台服务器进行抓取。这样，即使某台服务器出现问题，也不会影响其他服务器的正常工作。

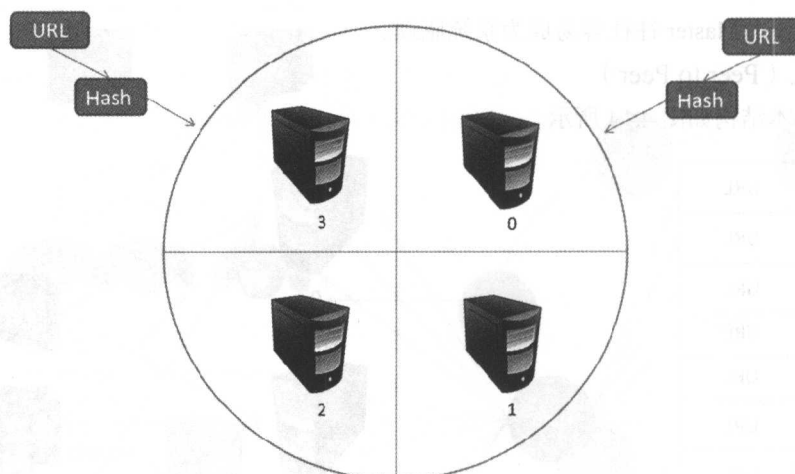


图 4.15

4.4.2 简单爬虫 Python 代码示例^①

总的来说，爬虫是一项非常成熟的技术。Python 提供了很好的类库，用 Python 实现一个简单的爬虫程序所需的代码非常少。下面给出一个简单的 Python 爬虫示例：

```
# -*- coding: utf-8 -*-

import urllib2
import urllib
import re
import time
#通过 url 获取网页源码 html
def getHtml(url):
    page = urllib2.urlopen(url)
    html = page.read()
    return html
#在 html 中找到匹配的 url
def getImg(html):
    #修改这里的匹配模式，适用于不同的网页
    reg = r'src="(http://.+?.jpg)" ' # +号后面加上? --->非贪婪模式
    imgre = re.compile(reg)
    imglist = re.findall(imgre,html)
    i = 0
    for imgurl in imglist:
        print imgurl
        urllib.urlretrieve(imgurl,'%s.jpg'%time.time()) #下载 imgurl 的图片并且用
        当前时间戳命名
        i+=1
    #return imglist
```

^① Python 实现简单爬虫功能示例，<http://www.cnblogs.com/fnng/p/3576154.html>。

```
url = "http://tieba.baidu.com/p/2772656630"
html = getHtml(url)
print getImg(html)
```

这里的核心是使用 `urllib.urlretrieve()` 方法直接将远程数据下载到本地。

上述代码通过一个 `for` 循环对获取的图片链接进行了遍历。为了使图片的文件名看上去更加规范，这里对其进行了重命名，命名规则为通过 `x` 变量加 1，保存的位置默认为程序的存放目录。

4.5 日志收集

任何一个生产系统在运行过程中都会产生大量的日志，日志往往隐藏了很多有价值的信息。在没有分析方法之前，这些日志存储一段时间后会就会被清理。随着技术的发展和分析能力的提高，日志的价值被重新重视起来。在分析这些日志之前，需要将分散在各个生产系统中的日志收集起来。本节介绍广泛应用的 Flume 日志收集系统。

4.5.1 Flume^①

1. 概述

Flume^②是 Cloudera 公司的一款高性能、高可用的分布式日志收集系统，现在已经是 Apache 的顶级项目。同 Flume 相似的日志收集系统还有 Facebook Scribe、Apache Chuwka。

2. Flume 发展历程

Flume 初始的发行版本目前被统称为 Flume OG (Original Generation)，属于 Cloudera。但随着 Flume 功能的扩展，Flume OG 代码工程臃肿、核心组件设计不合理、核心配置不标准等缺点逐渐暴露出来，尤其是在 Flume OG 的最后一个发行版本 0.94.0 中，日志传输不稳定现象尤为严重。为了解决这些问题，2011 年 10 月 22 日，Cloudera 完成了 Flume-728，对 Flume 进行了里程碑式的改动：重构核心组件、核心配置及代码架构，重构后的版本统称为 Flume NG (Next Generation)；改动的另一原因是将 Flume 纳入 Apache 旗下，Cloudera Flume 更名为 Apache Flume。

3. Flume 架构分析

1) 系统特点

① 可靠性

当节点出现故障时，日志能够被传送到其他节点上而不会丢失。Flume 提供了三种级别的可靠性保障，从强到弱依次为：`end-to-end`（收到数据后，Agent 首先将事件写到磁盘上，当数据传送成功后，再删除；如果数据发送失败，则重新发送）、`Store on Failure`（这也是 Scribe 采用的策略，当数

① 参考 <http://www.aboutyun.com/thread-7848-1-1.html> 官网用户手册，<http://flume.apache.org/FlumeUserGuide.html>。

② Github 地址 <https://github.com/apache/flume>。

据接收方崩溃时，将数据写到本地，待恢复后继续发送）、Best Effort（数据发送到接收方后，不会进行确认）。

② 可扩展性

Flume 采用了三层架构，分别为 Agent、Collector 和 Storage，每一层均可以水平扩展。其中，所有的 Agent 和 Collector 均由 Master 统一管理，这使得系统容易被监控和维护。并且 Master 允许有多个（使用 ZooKeeper 进行管理和负载均衡），这样就避免了单点故障问题。

③ 可管理性

当有多个 Master 时，Flume 利用 ZooKeeper 和 Gossip 保证动态配置数据的一致性。用户可以在 Master 上查看各个数据源或者数据流执行情况，并且可以对各个数据源进行配置和动态加载。Flume 提供了 Web 和 Shell Script Command 两种形式对数据流进行管理。

④ 功能可扩展性

用户可以根据需要添加自己的 Agent、Collector 或 Storage。此外，Flume 自带了很多组件，包括各种 Agent（如 File、Syslog 等）、Collector 和 Storage（如 File、HDFS 等）。

2) 系统架构

如图 4.16 所示是 Flume OG 的架构。

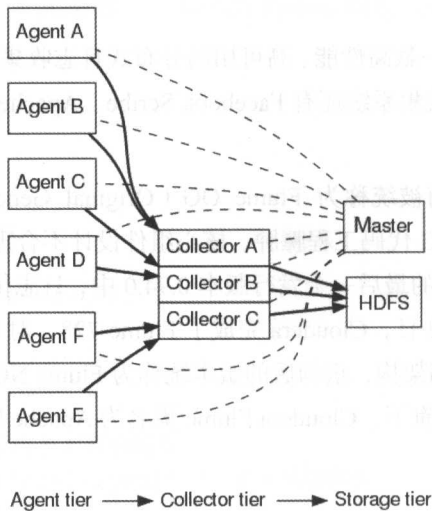


图 4.16

Flume NG 的架构如图 4.17 所示。

Flume 采用了分层架构，分别为 Agent、Collector 和 Storage。其中，Agent 和 Collector 均由 Source 和 Sink 两部分组成，Source 是数据来源，Sink 是数据去向。

Flume 使用了两个组件：Master 和 Node。Node 根据在 Master Shell 或 Web 中的动态配置，决定其是作为 Agent 还是作为 Collector。

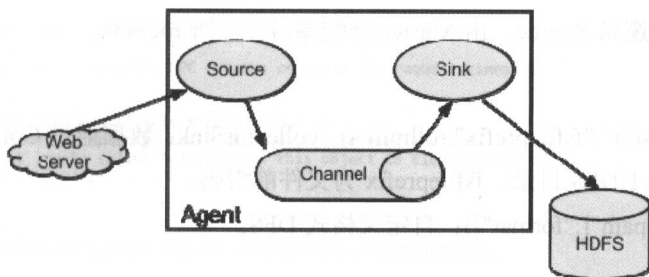


图 4.17

① Agent

Agent 的作用是将数据源的数据发送给 Collector。Flume 自带了很多直接可用的数据源 (Source)，如下。

- `text("filename")`: 将文件 `filename` 作为数据源，按行发送。
- `tail("filename")`: 探测 `filename` 新产生的数据，按行发送。
- `fsyslogTcp(5140)`: 监听 TCP 的 5140 端口，并将接收到的数据发送。
- `tailDir("dirname"[,fileregex=".*"[,startFromEnd=false[,recurseDepth=0]]])`: 监听目录中的文件末尾，使用正则表达式选定需要监听的文件（不包含目录），`recurseDepth` 为递归监听其下子目录的深度，同时提供了很多 Sink，如 `console(["format"])`，直接将数据显示在 console 上。
- `text("txtfile")`: 将数据写到文件 `txtfile` 中。
- `dfs("dfsfile")`: 将数据写到 HDFS 上的 `dfsfile` 文件中。
- `syslogTcp("host",port)`: 将数据通过 TCP 传递给 `host` 节点。
- `agentSink(["machine"[,port]])`: 等价于 `agentE2ESink`，如果省略 `machine` 参数，则默认使用 `flume.collector.event.host` 与 `flume.collector.event.port` 作为默认 collector。
- `agentDFOSink(["machine"[,port]])`: 本地热备 Agent。Agent 发现 Collector 节点故障后，不断检查 Collector 的存活状态以便重新发送 Event，在此期间产生的数据将缓存到本地磁盘中。
- `agentBESink(["machine"[,port]])`: 不负责的 Agent。如果 Collector 出现故障，将不作任何处理，它发送的数据也将被直接丢弃。
- `agentE2EChain`: 指定多个 Collector，以提高可用性。当向主 Collector 发送 Event 失效后，将转向第二个 Collector 发送；当所有的 Collector 都失效后，它还会再发送一遍。

② Collector

Collector 的作用是将多个 Agent 的数据汇总后，加载到 Storage 中。它的 Source 和 Sink 与 Agent 类似。

Source 如下。

- `collectorSource([port])`: Collector Source，监听端口汇聚数据。
- `autoCollectorSource`: 通过 Master 协调物理节点自动汇聚数据。

- logicalSource: 逻辑 Source, 由 Master 分配端口并监听 rpcSink。

Sink 如下。

- collectorSink("fsdir","fsfileprefix",rollmillis): collectorSink, 数据通过 Collector 汇聚之后发送到 HDFS, fsdir 是 HDFS 目录, fsfileprefix 为文件前缀码。
- customdfs("hdfspath":["format"]): 自定义格式 DFS。

③ Storage

Storage 是存储系统, 可以是一个普通 File, 也可以是 HDFS、Hive、HBase、分布式存储等。

④ Master

Master 负责管理、协调 Agent 和 Collector 的配置信息, 是 Flume 集群的控制器。

在 Flume 中, 最重要的抽象是 Data Flow (数据流)。Data Flow 描述了数据从产生、传输、处理到最终写入目标的一条路径, 如图 4.18 所示。

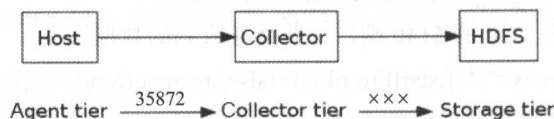


图 4.18

对于 Agent 数据流配置, 就是从哪里得到数据, 就把数据发送到哪个 Collector。

对于 Collector, 就是接收 Agent 发送过来的数据, 然后把数据发送到指定的目标机器上。

注: Flume 框架对 Hadoop 和 ZooKeeper 的依赖只存在于 JAR 包上, 并不要求 Flume 启动时必须将 Hadoop 和 ZooKeeper 服务同时启动。

3) 组件介绍

本文所说的 Flume 基于 1.4.0 版本。

① Client

路径: apache-flume-1.4.0-src\flume-ng-clients。

操作最初的数据, 把数据发送给 Agent。在 Client 与 Agent 之间建立数据沟通的方式有两种。

第一种方式: 创建一个 iclient 继承 Flume 已经存在的 Source, 如 AvroSource 或者 SyslogTcpSource, 但是必须保证所传输的数据 Source 可以理解。

第二种方式: 写一个 Flume Source 通过 IPC 或者 RPC 协议直接与已经存在的应用通信, 需要转换成 Flume 可以识别的事件。

Client SDK: 是一个基于 RPC 协议的 SDK 库, 可以通过 RPC 协议使应用与 Flume 直接建立连接。可以直接调用 SDK 的 api 函数而不用关注底层数据是如何交互的。

```

/**
 * <p>Send a single {@link Event} to the associated Flume source.</p>
 *
 * <p>This method blocks until the RPC returns or until the request times out.
 * </p>
 *
 * <p><strong>Note:</strong> If this method throws an
 * {@link EventDeliveryException}, there is no way to recover and the
 * application must invoke {@link #close()} on this object to clean up system
 * resources.</p>
 *
 * @param event
 * @return
 * @throws EventDeliveryException when an error prevents event delivery.
 */

```

```
public void append(Event event) throws EventDeliveryException;
```

```

/**
 * <p>Send a list of {@link PlainEvent} events to the associated Flume source.
 * </p>
 *
 * <p>This method blocks until the RPC returns or until the request times out.
 * </p>
 *
 * <p>It is strongly recommended that the number of events in the List be no
 * more than {@link #getBatchSize()}. If it is more, multiple RPC calls will
 * be required, and the likelihood of duplicate Events being stored will
 * increase.</p>
 *
 * <p><strong>Note:</strong> If this method throws an
 * {@link EventDeliveryException}, there is no way to recover and the
 * application must invoke {@link #close()} on this object to clean up system
 * resources.</p>
 *
 * @param events List of events to send
 * @return
 * @throws EventDeliveryException when an error prevents event delivery.
 */

```

```
public void appendBatch(List<Event> events) throws
    EventDeliveryException;
```

路径: apache-flume-1.4.0-src\flume-ng-sdk\src\main\java\org\apache\flume\api\RpcClient.java。

② NettyAvroRpcClient

Avro 是默认的 RPC 协议。NettyAvroRpcClient 和 ThriftRpcClient 分别对 RpcClient 接口进行了实现。

```

/**
 * Avro/Netty implementation of {@link RpcClient}.
 * The connections are intended to be opened before clients are given access so
 * that the object cannot ever be in an inconsistent when exposed to users.
 */
public class NettyAvroRpcClient extends AbstractRpcClient
    implements RpcClient {

    private ExecutorService callTimeoutPool;
    private final ReentrantLock stateLock = new ReentrantLock();

    /**
     * Guarded by {@code stateLock}
     */
    private ConnState connState;

    private InetSocketAddress address;
    private boolean enableSsl;
    private boolean trustAllCerts;
    private String truststore;
    private String truststorePassword;
    private String truststoreType;

    private Transceiver transceiver;
    private AvroSourceProtocol.Callback avroClient;
    private static final Logger logger = LoggerFactory
        .getLogger(NettyAvroRpcClient.class);
    private boolean enableDeflateCompression;
    private int compressionLevel;

```

路径: apache-flume-1.4.0-src\flume-ng-sdk\src\main\java\org\apache\flume\api\NettyAvroRpcClient.java。

```
public class ThriftRpcClient extends AbstractRpcClient {
    private static final Logger LOGGER =
        LoggerFactory.getLogger(ThriftRpcClient.class);

    private int batchSize;
    private long requestTimeout;
    private final Lock stateLock;
    private State connState;
    private String hostname;
    private int port;
    private ConnectionPoolManager connectionManager;
    private final ExecutorService callTimeoutPool;
    private final AtomicLong threadCounter;
    private int connectionPoolSize;
    private final Random random = new Random();

    public ThriftRpcClient() {
        stateLock = new ReentrantLock(true);
        connState = State.INIT;

        threadCounter = new AtomicLong(0);
        // OK to use cached threadpool, because this is simply meant to timeout
        // the calls - and is IO bound.
        callTimeoutPool = Executors.newCachedThreadPool(new ThreadFactory() {
```

路径: apache-flume-1.4.0-src\flume-ng-sdk\src\main\java\org\apache\flume\api\ThriftRpcClient.java。

使用 SDK 与 Flume 建立连接的实例如下:

```
import org.apache.flume.Event;
import org.apache.flume.EventDeliveryException;
import org.apache.flume.api.RpcClient;
import org.apache.flume.api.RpcClientFactory;
import org.apache.flume.event.EventBuilder;
import java.nio.charset.Charset;

public class MyApp {
    public static void main(String[] args) {
        MyRpcClientFacade client = new MyRpcClientFacade();
        // Initialize client with the remote Flume agent's host and port
        client.init("host.example.org", 41414);
        // Send 10 events to the remote Flume agent. That agent should be
        // configured to listen with an AvroSource.
        String sampleData = "Hello Flume!";
        for (int i = 0; i < 10; i++) {
            client.sendDataToFlume(sampleData);
        }
        client.cleanup();
    }
}

class MyRpcClientFacade {
    private RpcClient client;
    private String hostname;
    private int port;
    public void init(String hostname, int port) {
        // Setup the RPC connection
        this.hostname = hostname;
        this.port = port;
```

```

        this.client = RpcClientFactory.getDefaultInstance(hostname,port);
        // Use the following method to create a thrift client (instead of the above
line):
        // this.client = RpcClientFactory.getThriftInstance(hostname,port);
    }
    public void sendDataToFlume(String data) {
        // Create a Flume Event object that encapsulates the sample data
        Event event = EventBuilder.withBody(data,Charset.forName("UTF-8"));
        // Send the event
        try {
            client.append(event);
        } catch (EventDeliveryException e) {
            // clean up and recreate the client
            client.close();
            client = null;
            client = RpcClientFactory.getDefaultInstance(hostname,port);
            // Use the following method to create a thrift client (instead of the above
line):
            // this.client = RpcClientFactory.getThriftInstance(hostname,port);
        }
    }
    public void cleanUp() {
        // Close the RPC connection
        client.close();
    }
}

```

为了能够监听到关联端口，需要在配置文件中增加端口和 Host 配置信息。路径：apache-flume-

1.4.0-src\conf\flume-conf.properties.template。

```

client.type = default (for avro) or thrift (for thrift)
hosts = h1 # default client accepts only 1 host
           # (additional hosts will be ignored)
hosts.h1 = host1.example.org:41414 # host and port must both be specified
           # (neither has a default)
batch-size = 100 # Must be >=1 (default:100)
connect-timeout = 20000 # Must be >=1000 (default:20000)
request-timeout = 20000 # Must be >=1000 (default:20000)

```

除了以上两类实现外，FailoverRpcClient.java 和 LoadBalancingRpcClient.java 也分别对 RpcClient 接口进行了实现。

③ FailoverRpcClient

路径：apache-flume-1.4.0-src\flume-ng-sdk\src\main\java\org\apache\flume\api\FailoverRpcClient.java。

该组件主要实现了主备切换，采用<host>:<port>的形式，一旦当前连接失败，就会自动寻找下一个连接。

```

// Setup properties for the failover
Properties props = new Properties();

```



```

props.put("client.type","default_failover");
// List of hosts (space-separated list of user-chosen host aliases)
props.put("hosts","h1 h2 h3");
// host/port pair for each host alias
String host1 = "host1.example.org:41414";
String host2 = "host2.example.org:41414";
String host3 = "host3.example.org:41414";
props.put("hosts.h1",host1);
props.put("hosts.h2",host2);
props.put("hosts.h3",host3);
// create the client with failover properties
RpcClient client = RpcClientFactory.getInstance(props);
client.type = default_failover
hosts = h1 h2 h3                                # at least one is required,but 2 or
                                                    # more makes better sense

hosts.h1 = host1.example.org:41414
hosts.h2 = host2.example.org:41414
hosts.h3 = host3.example.org:41414
max-attempts = 3                                # Must be >=0 (default: number of hosts
                                                    # specified,3 in this case). A '0'
                                                    # value doesn't make much sense because
                                                    # it will just cause an append call to
                                                    # immediately fail. A '1' value means
                                                    # that the failover client will try only
                                                    # once to send the Event,and if it
                                                    # fails then there will be no failover
                                                    # to a second client,so this value
                                                    # causes the failover client to
                                                    # degenerate into just a default client.
                                                    # It makes sense to set this value to at
                                                    # least the number of hosts that you
                                                    # specified.

batch-size = 100                                # Must be >=1 (default:100)
connect-timeout = 20000                          # Must be >=1000 (default:20000)
request-timeout = 20000                          # Must be >=1000 (default:20000)

```

④ LoadBalancingRpcClient

该组件在有多个 Host 的时候起到负载均衡的作用。

```

// Setup properties for the load balancing
Properties props = new Properties();
props.put("client.type","default_loadbalance");
// List of hosts (space-separated list of user-chosen host aliases)
props.put("hosts","h1 h2 h3");
// host/port pair for each host alias
String host1 = "host1.example.org:41414";
String host2 = "host2.example.org:41414";
String host3 = "host3.example.org:41414";

```

```

props.put("hosts.h1",host1);
props.put("hosts.h2",host2);
props.put("hosts.h3",host3);
props.put("host-selector","random");// For random host selection
// props.put("host-selector","round_robin");// For round-robin host
// selection
props.put("backoff","true"); // Disabled by default.
props.put("maxBackoff","10000"); // Defaults 0,which effectively
// becomes 30000 ms

// Create the client with load balancing properties
RpcClient client = RpcClientFactory.getInstance(props);
client.type = default_loadbalance
hosts = h1 h2 h3 # At least 2 hosts are required
hosts.h1 = host1.example.org:41414
hosts.h2 = host2.example.org:41414
hosts.h3 = host3.example.org:41414
backoff = false # Specifies whether the client should
# back-off from (i.e. temporarily
# blacklist) a failed host
# (default:false).

maxBackoff = 0 # Max timeout in millis that a will
# remain inactive due to a previous
# failure with that host (default:0,
# which effectively becomes 30000)

host-selector = round_robin # The host selection strategy used
# when load-balancing among hosts
# (default:round_robin).
# Other values are include "random"
# or the FQCN of a custom class
# that implements
# LoadBalancingRpcClient$HostSelector

batch-size = 100 # Must be >=1 (default: 100)
connect-timeout = 20000 # Must be >=1000 (default: 20000)
request-timeout = 20000 # Must be >=1000 (default: 20000)

```

⑤ Embedded Agent

Flume 允许用户在自己的 Application 里内嵌一个 Agent。这个内嵌的 Agent 是一个轻量级的 Agent，不支持所有的 Source Sink Channel。

⑥ Transaction

Flume 的三个主要组件——Source、Sink、Channel 必须使用 Transaction 来进行消息收发。在 Channel 的类中会实现 Transaction 的接口，不管是 Source 还是 Sink，只要连接上 Channel，就必须先获取 Transaction 对象，如图 4.19 所示。

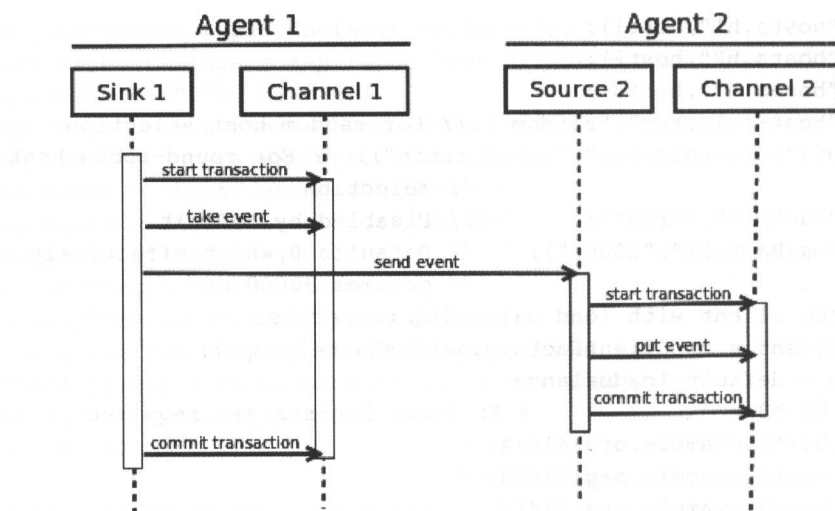


图 4.19

```

public interface Transaction {

    public enum TransactionState {Started, Committed, RolledBack, Closed };

    /**
     * <p>Starts a transaction boundary for the current channel operation. If a
     * transaction is already in progress, this method will join that transaction
     * using reference counting.</p>
     * <p><strong>Note</strong>: For every invocation of this method there must
     * be a corresponding invocation of {@linkplain #close()} method. Failure
     * to ensure this can lead to dangling transactions and unpredictable results.
     * </p>
     */
    public void begin();

    /**
     * Indicates that the transaction can be successfully committed. It is
     * required that a transaction be in progress when this method is invoked.
     */
    public void commit();

    /**
     * Indicates that the transaction can must be aborted. It is
     * required that a transaction be in progress when this method is invoked.
     */
    public void rollback();

    /**
     * <p>Ends a transaction boundary for the current channel operation. If a
     * transaction is already in progress, this method will join that transaction
     * using reference counting. The transaction is completed only if there
     * are no more references left for this transaction.</p>
     * <p><strong>Note</strong>: For every invocation of this method there must
     * be a corresponding invocation of {@linkplain #begin()} method. Failure
     * to ensure this can lead to dangling transactions and unpredictable results.
     * </p>
     */
    public void close();
} ? end Transaction ?

```

路径: apache-flume-1.4.0-src\flume-ng-core\src\main\java\org\apache\flume\Transaction.java。

Channel 对 Transaction 的实现如下:

```

public interface Channel extends LifecycleAware, NamedComponent {

    /**
     * <p>Puts the given event into the channel.</p>
     * <p><strong>Note</strong>: This method must be invoked within an active
     * {@link Transaction} boundary. Failure to do so can lead to unpredictable
     * results.</p>
     * @param event the event to transport.
     * @throws ChannelException in case this operation fails.
     * @see org.apache.flume.Transaction#begin()
     */
    public void put(Event event) throws ChannelException;

    /**
     * <p>Returns the next event from the channel if available. If the channel
     * does not have any events available, this method must return {@code null}.
     * </p>
     * <p><strong>Note</strong>: This method must be invoked within an active
     * {@link Transaction} boundary. Failure to do so can lead to unpredictable
     * results.</p>
     * @return the next available event or {@code null} if no events are
     * available.
     * @throws ChannelException in case this operation fails.
     * @see org.apache.flume.Transaction#begin()
     */
    public Event take() throws ChannelException;

    /**
     * @return the transaction instance associated with this channel.
     */
    public Transaction getTransaction();
} ? end Channel ?

```

路径: apache-flume-1.4.0-src\flume-ng-core\src\main\java\org\apache\flume\Channel.java。

一个实例如下:

```

Channel ch = new MemoryChannel();
Transaction txn = ch.getTransaction();
txn.begin();
try {
    Event eventToStage = EventBuilder.withBody("Hello Flume!", Charset.forName(
("UTF-8")));
    ch.put(eventToStage);
    txn.commit();
} catch (Throwable t) {
    txn.rollback();
    if (t instanceof Error) {
        throw (Error)t;
    }
} finally {
    txn.close();
}

```

⑦ Sink

Sink 的一个重要作用就是从 Channel 里获取事件, 然后把事件发送给下一个 Agent, 或者把事件存储到另外的仓库内。一个 Sink 会关联一个 Channel, 这是配置在 Flume 的配置文件的。SinkRunner.start()函数被调用后, 会创建一个线程, 该线程负责管理 Sink 的整个生命周期。Sink 需要实现 LifecycleAware 接口的 start()和 stop()方法。

- Sink.start(): 初始化 Sink, 设置 Sink 的状态, 可以进行事件收发。

- Sink.stop(): 进行必要的 cleanup 动作。
- Sink.process(): 负责具体的事件操作。

```
public class MySink extends AbstractSink implements Configurable {
    private String myProp;
    @Override
    public void configure(Context context) {
        String myProp = context.getString("myProp", "defaultValue");
        // Process the myProp value (e.g. validation)
        // Store myProp for later retrieval by process() method
        this.myProp = myProp;
    }
    @Override
    public void start() {
        // Initialize the connection to the external repository (e.g. HDFS) that
        // this Sink will forward Events to ..
    }
    @Override
    public void stop () {
        // Disconnect from the external repository and do any
        // additional cleanup (e.g. releasing resources or nulling-out
        // field values) ..
    }
    @Override
    public Status process() throws EventDeliveryException {
        Status status = null;
        // Start transaction
        Channel ch = getChannel();
        Transaction txn = ch.getTransaction();
        txn.begin();
        try {
            // This try clause includes whatever Channel operations you want to do
            Event event = ch.take();
            // Send the Event to the external repository.
            // storeSomeData(e);
            txn.commit();
            status = Status.READY;
        } catch (Throwable t) {
            txn.rollback();
            // Log exception, handle individual exceptions as needed
            status = Status.BACKOFF;
            // re-throw all Errors
            if (t instanceof Error) {
                throw (Error)t;
            }
        } finally {
            txn.close();
        }
        return status;
    }
}
```

```

}
}

```

⑧ Source

Source 的作用是从 Client 端接收事件,然后把事件存储到 Channel 中。PollableSourceRunner.start() 用于创建一个线程,管理 PollableSource 的生命周期。同样也需要实现 start()和 stop()两种方法。需要注意的是,还有一类 Source,被称为 EventDrivenSource。区别是 EventDrivenSource 有自己的回调函数用于捕捉事件,并不是每个线程都会驱动一个 EventDrivenSource。

以下是一个 PollableSource 的例子:

```

public class MySource extends AbstractSource implements Configurable,
PollableSource {
    private String myProp;
    @Override
    public void configure(Context context) {
        String myProp = context.getString("myProp", "defaultValue");
        // Process the myProp value (e.g. validation, convert to another type, ...)
        // Store myProp for later retrieval by process() method
        this.myProp = myProp;
    }
    @Override
    public void start() {
        // Initialize the connection to the external client
    }
    @Override
    public void stop () {
        // Disconnect from external client and do any additional cleanup
        // (e.g. releasing resources or nulling-out field values) ..
    }
    @Override
    public Status process() throws EventDeliveryException {
        Status status = null;
        // Start transaction
        Channel ch = getChannel();
        Transaction txn = ch.getTransaction();
        txn.begin();
        try {
            // This try clause includes whatever Channel operations you want to do
            // Receive new data
            Event e = getSomeData();
            // Store the Event into this Source's associated Channel(s)
            getChannelProcessor().processEvent(e)
            txn.commit();
            status = Status.READY;
        } catch (Throwable t) {
            txn.rollback();
            // Log exception, handle individual exceptions as needed
            status = Status.BACKOFF;

```



```
// re-throw all Errors
if (t instanceof Error) {
    throw (Error)t;
}
} finally {
    txn.close();
}
return status;
}
}
```

4) Flume 使用模式^①

(1) 多 Agent 串联，如图 4.20 所示。

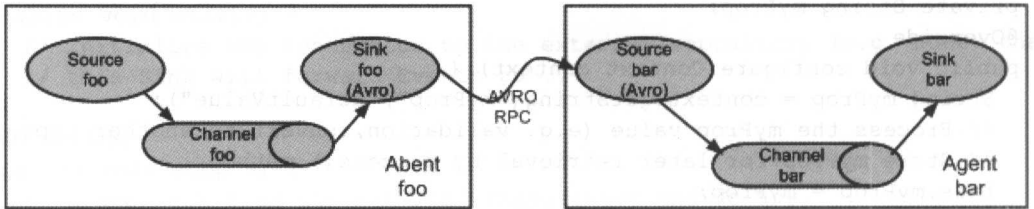


图 4.20

(2) 多 Agent 合并，如图 4.21 所示。

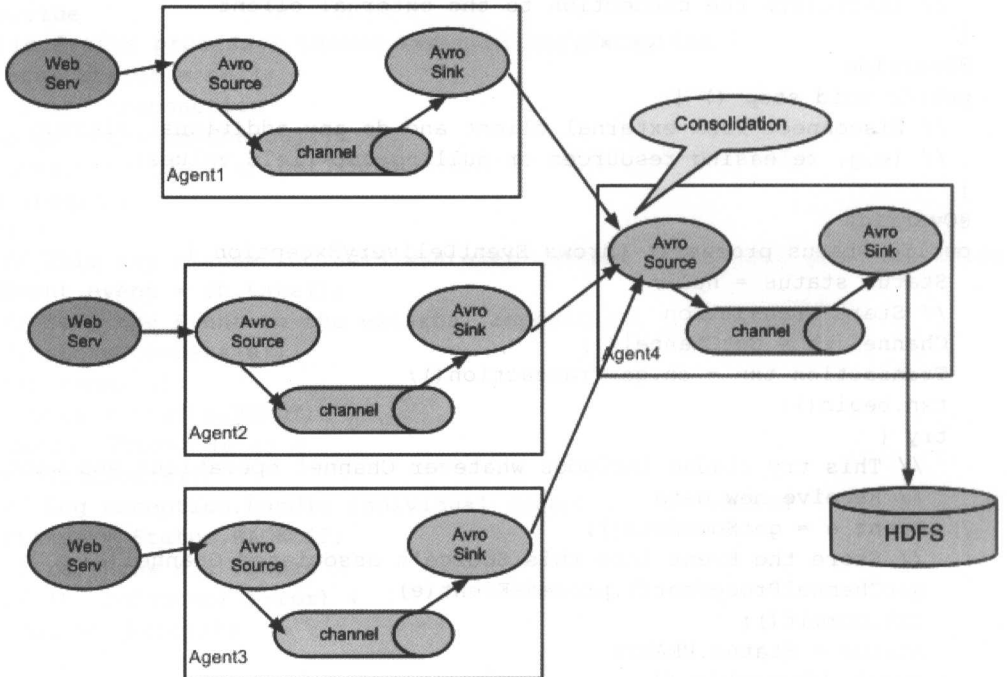


图 4.21

① 参考 <http://flume.apache.org/FlumeUserGuide.html>。

(3) 单 Source 的多种处理, 如图 4.22 所示。

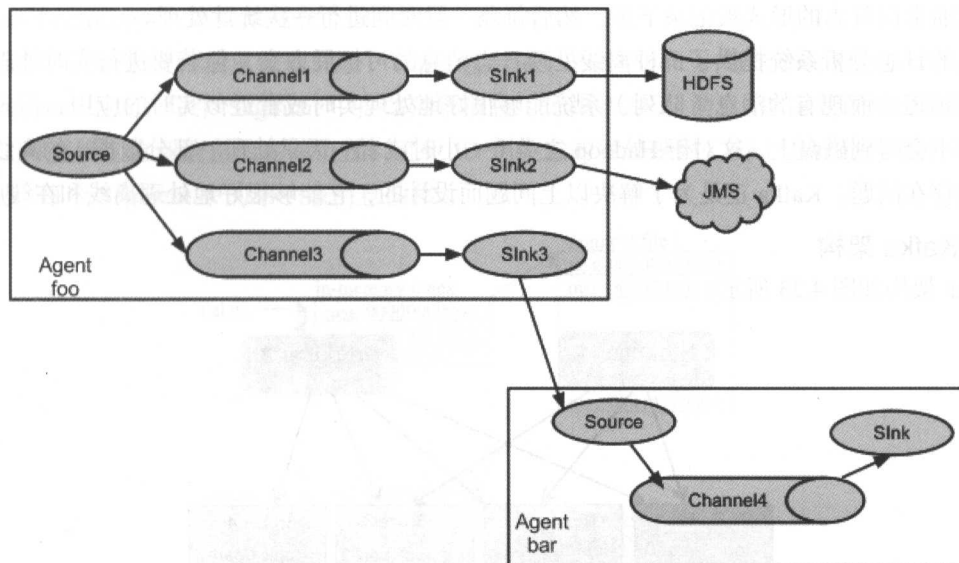


图 4.22

4.5.2 其他日志收集组件

Scribe 和 Chukwa 也是不错的日志收集组件。

- Scribe 主页: <https://github.com/facebook/scribe>。
- Chukwa 主页: <http://incubator.apache.org/chukwa/>。

4.6 数据分发中间件

4.6.1 数据分发中间件的作用

数据采集上来后, 需要送到后端的组件进行进一步的分析, 前端的采集和后端的处理往往是多对多的关系。为了简化传送逻辑、增强灵活性, 在前端的采集和后端的处理之间需要一个消息中间件来负责消息转发, 以保障消息可靠性, 匹配前后端的速度差。

4.6.2 Kafka 架构和原理^①

1. Kafka 产生背景

Kafka 是 LinkedIn 于 2010 年 12 月开源的消息系统, 主要用于处理活跃的流式数据。活跃的流式

^① Kafka 相关资料: <http://blog.cloudera.com/blog/2014/11/flafka-apache-flume-meets-apache-kafka-for-event-processing/> 和 <http://blog.cloudera.com/blog/2015/08/inside-santanders-near-real-time-data-ingest-architecture/>。

数据在 Web 网站应用中很常见,这些数据包括网站的 PV、用户访问了什么内容、搜索了什么内容等。这些数据通常以日志的形式被记录下来,然后每隔一段时间进行一次统计处理。

传统的日志分析系统提供了一种离线处理日志消息的可扩展方案,但若要进行实时处理,通常会有较大延迟。而现有的消息(队列)系统能够很好地处理实时或者近似实时的应用,但未处理的数据通常不会写到磁盘上,这对于 Hadoop 之类(一小时或者一天只处理一部分数据)的离线应用而言,可能存在问题。Kafka 正是为了解决以上问题而设计的,它能够很好地处理离线和在线应用。

2. Kafka 架构

Kafka 架构如图 4.23 所示。

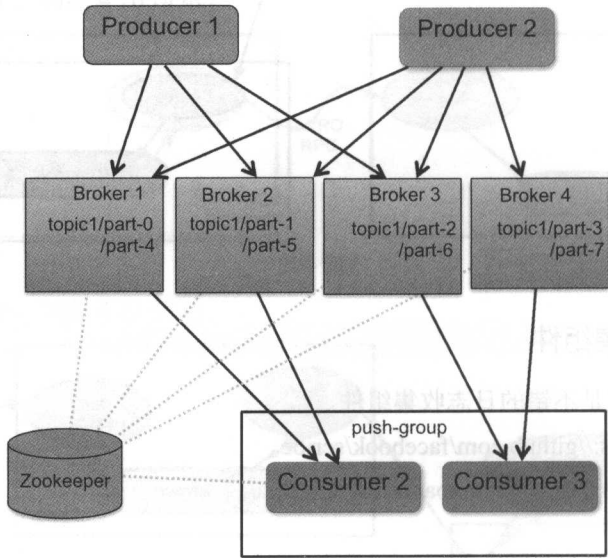


图 4.23

整个架构中包括三个角色。

- 生产者 (Producer): 消息和数据产生者。
- 代理 (Broker): 缓存代理, Kafka 的核心功能。
- 消费者 (Consumer): 消息和数据消费者。

整体架构很简单,Kafka 给 Producer 和 Consumer 提供注册的接口,数据从 Producer 发送到 Broker, Broker 承担一个中间缓存和分发的作用,负责分发注册到系统中的 Consumer。

3. 设计要点

Kafka 非常高效,下面介绍一下 Kafka 高效的原因,对理解 Kafka 非常有帮助。

(1) 直接使用 Linux 文件系统的 Cache 来高效缓存数据。

(2) 采用 Linux Zero-Copy 提高发送性能。传统的数据发送需要发送 4 次上下文切换,采用 Sendfile 系统调用之后,数据直接在内核态交换,系统上下文切换减少为 2 次。根据测试结果,可以

息的时候，偏移量也线性增加。但是实际偏移量由消费者控制，消费者可以重置偏移量，以重新读取消息。

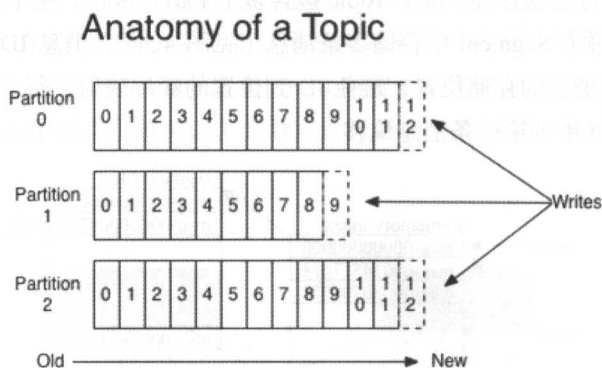


图 4.25

可以看到，这种设计对消费者来说操作自如，一个消费者的操作不会影响其他消费者对此 Log 的处理。

再来说说分区。Kafka 中采用分区的设计有两个目的：一是可以处理更多的消息，而不受单台服务器的限制，Topic 拥有多个分区，意味着它可以不受限制地处理更多的数据；二是分区可以作为并行处理的单元。

Kafka 会为每个分区创建一个文件夹，文件夹的命名方式为 topicName-分区序号，如图 4.26 所示。

```

pw@linux:/home/pw/nmon_stat> ll /home/pw/kafka/kafka_log/
total 36
-rw-r--r-- 1 root root 81 2015-10-19 16:06 recovery-point-offset-checkpoint
-rw-r--r-- 1 root root 116 2015-10-19 16:07 replication-offset-checkpoint
drwxr-xr-x 2 root root 4096 2015-10-19 15:59 test10-0
drwxr-xr-x 2 root root 4096 2015-10-19 15:59 test10-2
drwxr-xr-x 2 root root 4096 2015-10-19 15:59 test10-3
drwxr-xr-x 2 root root 4096 2015-10-19 15:59 test10-4
drwxr-xr-x 2 root root 4096 2015-10-19 15:59 test10-6
drwxr-xr-x 2 root root 4096 2015-10-19 15:59 test10-8
drwxr-xr-x 2 root root 4096 2015-10-19 15:59 test10-9

```

图 4.26

而分区是由多个 Segment 组成的，是为了方便进行日志清理、恢复等工作。每个 Segment 以该 Segment 第一条消息的 offset 命名并以“.log”作为后缀。另外还有一个索引文件，它标明了每个 Segment 下包含的 Log Entry 的 offset 范围，文件命名方式也是如此，以“.index”作为后缀，如下：

```
00000000000000000000000000000000.index
00000000000000000000000000000000.log
0000000000000000000000368769.index
0000000000000000000000368769.log
0000000000000000000000737337.index
0000000000000000000000737337.log
00000000000000000000001105814.index
00000000000000000000001105814.log
.....
```

我们再来看看索引和日志文件内部的关系是什么，如图 4.27 所示。

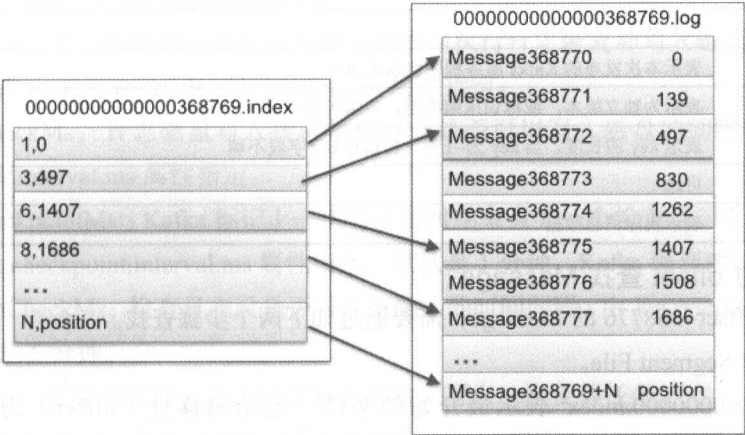


图 4.27

索引文件存储大量元数据，数据文件存储大量消息（Message），索引文件中的元数据指向对应数据文件中 Message 的物理偏移地址。以索引文件中的元数据 3,497 为例，依次在数据文件中表示第三个 Message（在全局 Partition 中表示第 368 772 个 Message），以及该消息的物理偏移地址为 497。

Segment 的 Log 文件由多个 Message 组成，下面详细说明 Message 的物理结构，如图 4.28 所示。

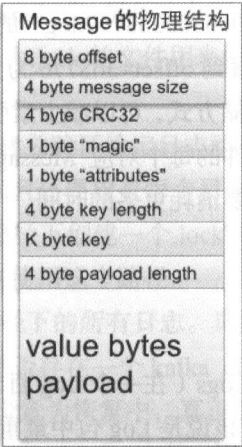


图 4.28

参数说明如表 4.2 所示。

表 4.2

关 键 字	解 释 说 明
8 byte offset	在 Partition（分区）内的每条消息都有一个有序的 ID，这个 ID 被称为偏移（offset），它可以唯一确定每条消息在 Partition（分区）内的位置。即 offset 表示 Partition 的第多少个 Message
4 byte message size	Message 的大小

续表

关键字	解释说明
4 byte CRC32	用 CRC32 校验 Message
1 byte "magic"	表示本次发布的 Kafka 服务程序协议版本号
1 byte "attributes"	表示为独立版本，或标识压缩类型，或编码类型
4 byte key length	表示 key 的长度。当 key 为-1 时，K byte key 字段不填
K byte key	可选
value bytes payload	表示实际消息数据

5. 如何通过 offset 查找 Message

例如，读取 offset=368776 的 Message，需要通过如下两个步骤查找。

第一步：查找 Segment File。

00000000000000000000.index 表示最开始的文件，起始偏移量（offset）为 0；第二个文件 00000000000000000000368769.index 的起始偏移量为 368770（368769 + 1）；同样，第三个文件 00000000000000000000737337.index 的起始偏移量为 737338（737337 + 1），依此类推。以起始偏移量命名并排序这些文件，只要根据 offset**二分查找**文件列表，就可以快速定位到具体文件。

当 offset=368776 时，定位到 00000000000000000000368769.index|log。

第二步：通过 Segment File 查找 Message。

通过第一步定位到 Segment File，当 offset=368776 时，依次定位到 00000000000000000000368769.index 的元数据物理位置和 00000000000000000000368769.log 的物理偏移地址，然后再通过 00000000000000000000368769.log 顺序查找，直到 offset=368776 为止。

Segment Index File 采取稀疏索引存储方式，可以减少索引文件大小，通过 Linux mmap 接口可以直接进行内存操作。稀疏索引为数据文件的每个对应 Message 设置一个元数据指针，它比稠密索引节省了更多的存储空间，但查找起来需要消耗更多的时间。

6. 主要代码解读

1) Log 包相关组件

① LogManager

LogManager 管理 Broker 上所有的 Logs（在一个 Log 目录下），一个 Topic 的一个 Partition 对应一个 Log（一个 Log 子目录）。这个类应该说是 Log 包中最重要的类，也是 Kafka 日志管理子系统的入口。日志管理器（Log Manager）负责创建日志、获取日志、清理日志。所有的日志读/写操作都交给具体的日志实例来完成。日志管理器维护多个路径下的日志文件，并且会自动比较不同路径下的文件数目，然后选择在最少的日志路径下创建新的日志。Log Manager 不会尝试去移动分区，另外专门有一个后台线程定期裁剪过量的日志段。下面来看看这个类的构造函数参数。

（1）logDirs：Log Manager 管理的多组日志目录。

（2）topicConfigs：topic=>topic 的 LogConfig 的映射。

(3) defaultConfig: 一些全局性的默认日志配置。

(4) cleanerConfig: 日志压缩清理的配置。

(5) ioThreads: 每个数据目录都可以创建一组线程执行日志恢复和写入磁盘, 这个参数就是这组线程的数目, 由 num.recovery.threads.per.data.dir 属性指定。

(6) flushCheckMs: 日志磁盘写入线程检查日志是否可以写入磁盘的间隔, 默认是毫秒, 由 log.flush.scheduler.interval.ms 属性指定。

(7) flushCheckpointMs: Kafka 标记上一次写入磁盘结束点为一个检查点, 用于日志恢复的间隔, 由 log.flush.offset.checkpoint.interval.ms 属性指定, 默认是 1 分钟, Kafka 强烈建议不要修改此值。

(8) retentionCheckMs: 检查日志段是否可以被删除的时间间隔, 由 log.retention.check.interval.ms 属性指定, 默认是 5 分钟。

(9) scheduler: 任务调度器, 用于指定日志删除、写入、恢复等任务。

(10) brokerState: Kafka Broker 的状态类(在 kafka.server 包中)。Broker 的状态默认有未运行(not running)、启动中(starting)、从上次未正常关闭恢复中(recovering from unclean shutdown)、作为 Broker 运行中(running as broker)、作为 Controller 运行中(running as controller)、挂起中(pending), 以及关闭中(shutting down)。当然, Kafka 允许自定制状态。

(11) time: 和很多类的构造函数参数一样, 是提供时间服务的变量。Kafka 在恢复日志的时候是借助检查点文件来进行的, 因此每个需要进行日志恢复的路径下都需要有这样一个检查点文件, 名称固定为“recovery-point-offset-checkpoint”。另外, 由于在执行一些操作时需要将目录下的文件锁住, 因此, Kafka 还创建了一个扩展名为.lock 的文件用来标识这个目录当前是被锁住的。

下面针对具体的方法一一分析。

(1) createAndValidateLogDirs: 创建并验证给定日志路径的合法性, 特别要保证不能出现重复路径, 并且要创建那些不存在的路径, 而且还要检查每个目录都是可读的。

(2) lockLogDirs: 在给定的所有路径下创建一个.lock 文件。如果某个路径下已经有.lock 文件, 则说明 Kafka 的另一个进程或线程正在使用这个路径。

(3) loadLogs: 恢复并加载给定路径下的所有日志。具体做法是为每个路径创建一个线程池。为了向后兼容, 该方法要在路径下寻找是否存在一个.kafka_cleanShutdown 文件, 如果存在的话就跳过这个恢复阶段, 否则将 Broker 的状态设置为恢复中, 真正的恢复工作是由 Log 实例来完成的。然后读取对应路径下的 recovery-point-offset-checkpoint 文件, 读出要恢复的检查点。前面提到检查点文件的格式大概类似于下面的内容:

第一行必须是版本 0; 第二行是 Topic/分区数; 以下每行都有三个字段, 即 Topic、Partition、Offset。读完这个文件后, 会创建一个 TopicAndPartition => offset 的 Map。

```
=====
0
9
log-topic 0 48
```

```

kafkatopic 1 0
abcd 1 0
abcd 0 0
autocreated 0 2
abc 0 0
accesslog_topic 0 0
kafkatopic 0 0
autocreated 1 1
=====

```

之后为每个目录下的子目录都构建一个 Log 实例，然后使用线程池调度执行清理任务，最后删除这些任务对应的 cleanShutdown 文件。至此，日志加载过程结束。

(4) startup: 开启后台线程进行日志冲刷 (flush) 和日志清理。主要使用调度器安排 3 个调度任务: cleanupLogs、flushDirtyLogs 和 checkpointRecoveryPointOffsets——自然有 3 个对应的实现方法。同时判断是否启用了日志压缩，如果启用了，则调用 Cleaner 的 startup 方法开启日志清理，如下：

```

def startup() {
    /* Schedule the cleanup task to delete old logs */
    if(scheduler != null) {
        info("Starting log cleanup with a period of %d ms.".format(retentionCheckMs))
        scheduler.schedule("kafka-log-retention",
                           cleanupLogs,
                           delay = InitialTaskDelayMs,
                           period = retentionCheckMs,
                           TimeUnit.MILLISECONDS)
        info("Starting log flusher with a default period of %d ms.".format(
flushCheckMs))
        scheduler.schedule("kafka-log-flusher",
                           flushDirtyLogs,
                           delay = InitialTaskDelayMs,
                           period = flushCheckMs,
                           TimeUnit.MILLISECONDS)
        scheduler.schedule("kafka-recovery-point-checkpoint",
                           checkpointRecoveryPointOffsets,
                           delay = InitialTaskDelayMs,
                           period = flushCheckpointMs,
                           TimeUnit.MILLISECONDS)
    }
    if(cleanerConfig.enableCleaner)
        cleaner.startup()
}

```

(5) shutdown: 关闭所有日志。首先关闭所有清理者线程，然后为每个日志目录创建一个线程池，执行目录下日志文件的写入磁盘与关闭操作，同时更新外层文件中检查点文件的对应记录。

(6) logsByTopicPartition: 返回一个 Map 保存 TopicAndPartition => Log 的映射。

(7) allLogs: 返回所有 Topic 分区的日志。

(8) logsByDir: 日志路径 => 路径下所有日志的映射。

- (9) `flushDirtyLogs`: 将任何超过写入间隔且有未写入消息的日志全部冲刷到磁盘上。
- (10) `checkpointLogsInDir`: 在给定的路径中标记一个检查点。
- (11) `checkpointRecoveryPointOffsets`: 将日志路径下所有日志的检查点写入一个文本文件中 (`recovery-point-offset-checkpoint`)。
- (12) `truncateTo`: 截断分区日志到指定的 `offset`, 并使用这个 `offset` 作为新的检查点 (恢复点)。具体做法就是遍历给定的 `Map` 集合, 获取对应分区的日志, 如果要截断的 `offset` 比该日志当前正在使用的日志段的基础位移小 (也就是说要截断一部分当前日志段), 则需要暂停清理者线程。之后开始执行阶段操作, 最后再恢复清理者线程。
- (13) `tuncateFullyAndStartAt`: 删除一个分区所有的数据并在新 `offset` 处开启日志。操作前后分别需要暂停和恢复清理者线程。
- (14) `getLog`: 返回某个分区的日志。
- (15) `createLog`: 为给定分区创建一个新的日志。如果日志已经存在, 则返回。
- (16) `deleteLog`: 删除一个日志。
- (17) `nextLogDir`: 创建日志时选择下一个路径。目前的实现是计算每个路径下的分区数, 然后选择最少的那个。
- (18) `cleanupExpiredSegments`: 删除那些过期的日志段, 也就是当前时间减去最近修改时间超出规定的那些日志段, 并且返回被删除日志段的个数。
- (19) `cleanupSegmentsToMaintainSize`: 如果没有设定 `log.retention.bytes`, 则直接返回 0, 表示不需要清理任何日志段 (这也是默认情况, 因为 `log.retention.bytes` 默认是 -1); 否则计算出该属性值与日志大小的差值。如果这个差值能够容纳某个日志段的大小, 那么这个日志段就需要被删除。
- (20) `cleanupLogs`: 删除所有满足条件的日志, 返回被删除的日志数。

② Log

日志类, 笔者认为其是这个包中最重要的两个类之一 (另一个是 `LogManager`), 以伴生对象的方式提供。先说 `Log Object`, 既然是 `Object`, 就定义了一些类级别的变量, 如定义了一个日志文件的扩展名是 `.log`; 索引文件的扩展名是 `.index`; 要被删除的文件的扩展名是 `.deleted`; 要被执行日志清理的临时文件的扩展名是 `.cleaned`; 在做 `swap` 过程中的临时文件的扩展名是 `.swap`。还有一个扩展名 `.kafka_cleanshutdown`, 在 `Kafka 0.8.2` 版本中已经被禁用。除了这些扩展名外, 这个 `Object` 还定义了一些常用的方法。

- (1) `filenamePrefixFromOffset`: 其实就是为给定的 `Offset` 前面补至 20 位, 生成日志段文件名, 用于文件名排序 (使用 `ls` 命令)。
- (2) `logFilename`: 使用给定的基础 `Offset` 在给定的路径下生成对应的日志文件。比如位移是 1, 那么生成的文件名是 `0...` (总共 19 个) `1.log`。
- (3) `indexFilename`: 与 `logFilename` 类似, 只不过生成的索引文件名是 `0...` (共 19 个) `1.index`。
- (4) `parseTopicPartitionName`: 将日志所在路径的名称解析成 `Topic`+分区封装到一个 `TopicPartition`

对象返回。比如，路径名字是 log-topic-0，那么 Topic 名字就是 log-topic，分区号就是 0。

说完了 Log Object，再来说说 Log Class。这里要说的日志只能在尾部追加消息，一个日志对象就是一组日志段 (LogSegment) 对象，每个日志段都有一个基础 offset 标识该日志段中第一条消息的位置。Kafka 支持基于大小和时间的规则创建新的日志段。

Log 类有 5 个构造器参数，如下。

- (1) dir: 日志段被创建在哪个目录下。
- (2) config: 日志配置信息。
- (3) recoveryPoint: 恢复的起始 offset，即尚未被写入磁盘的第一个 offset。
- (4) scheduler: 用于后台操作的一个调度器线程池，主要用于异步删除日志段和在日志段被切分时使用。

(5) time: 提供时间服务的对象实例。

该类被标记为线程安全的，因此需要一个锁对象来保护对日志的并发修改。同时定义了一个字段 lastflushedTime，专门保存该日志上次被写入磁盘的时间。前面说过，一个日志对象是由多个日志段组成的，所以该类定义了一组日志段 Map 对象 (Key 就是日志段的基础 offset，Value 就是该日志段) 表示该日志包含的所有日志段: Segments。值得注意的是，这个 Map 对象使用了 java.concurrent 包中的 ConcurrentNavigableMap 类，该类提供了很多对于 Map 的便捷的导航方法。另外，Log 类还定义了一个 nextOffsetMetadata 变量用于计算下一条消息的位移，主要计算方法就是调用 LogSegment 的 nextOffset 方法。同时，根据给定的路径名，该类还定义了一个 TopicAndPartition 对象，把从路径名中提取出来的 Topic 和分区信息保存起来，表明这个日志是属于哪个 Topic、哪个分区的。最后，该类定义了 4 个度量元，分别统计日志的起始位移、结束位移、日志大小及日志段的数目。

分析完所有的类成员字段，下面对类定义的方法进行逐一分析。

(1) name: 返回日志路径名称。

(2) updateLogEndOffset: 使用给定的 offset 创建一个新的 LogOffsetMetadata 对象，并更新到 nextOffsetMetadata 变量。

(3) hasCleanShutdownFile: 判断是否存在 clean shutdown 文件——这个功能在 Kafka 0.8.2 版本以后就不再使用了，主要是为了后向兼容。

(4) numberOfSegments: 日志的日志段的数目——主要调用 Map.size 方法实现，该方法的时间复杂度是 $O(n)$ ，所以在日志段数目比较多时要慎用。

(5) close: 以同步的方式将 Map 中的所有日志段都关闭。

(6) size: 所有日志段的字节数总和。

(7) logStartOffset: 日志段集合中第一个日志段的基础位移，也就是这个日志对象的基础位移。

(8) logEndOffsetMetadata: 下一条将要被加入日志的消息的位移元数据，直接返回 nextOffsetMetadata 字段。

(9) logEndOffset: 下一条将要被加入日志的消息的位移。

- (10) `unflushedMessage`: 已加入日志但未写入磁盘的消息数。
- (11) `flush`: 写入所有日志段中尚未持久化的消息。
- (12) `delete`: 完全删除该日志文件及目录, 并清空日志段 Map。
- (13) `lastFlushTime`: 返回最近一次写入磁盘的时间。
- (14) `activeSegment`: 日志段 Map 中的最后一个日志段表示当前活跃的日志段。
- (15) `logSegments` 无参版: 返回按照 `offset` 递增排序的日志段集合。
- (16) `addSegment`: 将给定的日志段对象加入现有的日志段 Map 中。
- (17) `asyncDeleteSegment`: 异步删除给定的日志段对象, 通常是请求发起 1 分钟之后开始执行。
- (18) `replaceSegments`: 该方法只在启用了日志清理 (Log Cleaning) 时才会被用到。主要逻辑就是将 `cleaned` 文件转换成 `swap` 文件, 然后再将 `swap` 文件加入现有的日志段 Map 中。之后遍历整个要删除的日志段集合, 如果要删除的日志段的基础位移与要新加的日志段的基础位移不一样, 则说明该日志段的确要被删除, 那么调用异步删除日志段的方法将其删除。遍历之后, 再删除新加的日志段的扩展名 `.swap`。
- (19) `deleteSegment`: 异步删除日志段, 在日志段 Map 中将日志段记录去除, 并且将对应的索引文件重命名为 `*.deleted`。
- (20) `logSegments` 带参版: 首先获取不大于 `from` 的最大位移, 如果不存在这样的 `offset`, 那么直接返回小于 `to` 的所有位移对应的日志段; 否则返回 `[from,to]` 范围内的日志段。
- (21) `truncateFullyAndStartAt`: 删除日志中的所有数据, 创建一个空的日志段并设置新的 `offset` (设置起始 `offset` 和结束 `offset` 为 `newOffset`), 最后更新恢复点位移。
- (22) `truncateTo`: 将日志截断, 使之保存的最大 `offset` 不会超过给定的 `targetOffset`。当然, 如果 `targetOffset` 已经比现有日志的结束位移大, 那么自然什么都不用做。另外, 在截断的过程中, 还需要判断该 Log 的最小位移 (也就是第一个日志段的基础位移), 如果比 `targetOffset` 大, 那么直接调用 `truncateFullyAndStartAt` 方法删除所有日志数据并设置新的位移点; 否则逐一删除那些起始位移比 `targetOffset` 大的日志段。此时 `activeSegment` 会自动变成当前删除之后最新的那个日志段, 所以还要对 `activeSegment` 进行截断操作。之后更新下一条消息 `offset`, 并重设恢复点位移。
- (23) `roll`: 日志段的切分。使用当前结束位移作为新日志段的起始位移, 并把新日志段加入日志段 Map 中, 然后发起一个异步调度任务将旧有日志段的数据同步到磁盘上, 最后返回新创建的日志段。
- (24) `maybeRoll`: 也是做日志段的切分, 不过是有条件的: 日志段已经满了; 已过最大时间; 索引文件已经满了。三个条件满足其一就会触发切分操作。
- (25) `deleteOldSegments`: 删除那些满足条件的日志段 (所谓的条件无非就是满足大小或时间方面的要求), 返回删除日志段的个数。具体做法是: 筛选出那些同时满足给定条件并且不是当前激活日志段或大小不为空的日志段, 然后比较一下, 看看要删除的是否是所有日志段, 如果是, 则直接

调用 `roll` 方法进行切分（因为 Kafka 至少要保留一个日志段）；否则直接遍历该候选日志段集合，然后删除之。

（26）`convertToOffsetMetadata`：将给定的位移转换成对应的 `LogOffsetMetadata` 对象。该方法主要用于副本管理。

（27）`read`：先说说这个方法的返回对象 `FetchDataInfo`——这是一个 `case` 类，包含了日志位移元数据信息及一个消息集合。这个方法也很简单，就是从日志中读取消息，将起始位移和读取到的消息集合封装进一个 `FetchDataInfo` 中。此方法接收 3 个参数：`startOffset` 表示读取操作执行的开始位移点；`maxLength` 表示最多读取的字节数；`maxOffset` 表示读取操作不能超过的位移点，即返回的消息集合中不能包含该位移。具体逻辑如下：首先检查下一条消息的位移与给定的起始位移，如果二者相等，则直接返回；否则，找到小于等于 `startOffset` 的最大位移所在的日志段。如果 `startOffset` 比当前最大位移还大，或者根本没有找到刚才的日志段，则说明要读取的内容已经超出日志当前的结束 `offset`，直接报错退出。如果到这里运行正常的话，那么下面就开始循环读取消息：如果读取的消息集合不为空，则直接返回；否则跳到下一个日志段继续读取，直到下一个日志段为空则退出循环。

（28）`trimInvalidBytes`：消除消息集合尾部的无效字节。在学习这个方法之前，首先要了解 Kafka 在这个 `scala` 文件中定义的一个 `case class` 类：`LogAppendInfo` 类。该类保存了每个消息集合的各种信息，包括这个集合的起始位移、结束位移、未压缩消息（`shallow message`）数、合法字节数、用到的压缩算法，以及表明该消息集合位移是否是单调增加的布尔值。现在再来看 `trimInvalidBytes` 方法，这个方法接收两个参数：一个是要做 `trim` 的消息集合；另一个是用 `LogAppendInfo` 对象标识的消息集合的通用信息。返回的结果自然是被 `trim` 过的消息集合（可能与原消息集合相同）。具体做法是：首先计算出这个消息集合的合法字节数（这要通过 `analyzeAndValidateMessageSet` 方法给出，后面会说明这个方法），如果合法字节数小于 0，则直接报错退出。如果该字节数就是消息集合的字节数，则说明不用做 `trim` 直接返回传入的 `Messages` 即可；否则说明有非法字节，那么就新建一个 `ByteBufferMessageSet` 消息集合，将 `limit` 设置为刚才计算的合法字节数，然后返回。

（29）`analyzeAndvalidateMessageSet`：上面的方法提到了一个消息集合的合法字节与非法字节，那么，如何定义合法性呢？答案就由这个方法给出。从字面上来说，这个方法做的工作就是分析验证消息集合，主要包括：a. 验证消息 CRC 码；b. 验证消息长度的合法性；c. 计算消息集合的起始位移；d. 计算结束位移；e. 消息集合中的消息数；f. 计算合法字节数（合法消息字节数的累加和）；g. 验证位移是否单调增加；h. 验证是否使用了压缩，如果指定了多个，则只以最后一条消息的压缩算法为准。

（30）`loadSegments`：该方法用于加载磁盘上的日志文件。具体逻辑如下：a. 如果给定的路径不存在，则创建出来。b. 遍历该目录路径下的所有文件，删除那些临时文件（扩展名为 `.deleted` 和 `.cleaned` 的文件）。c. 如果发现是以 `.swap` 结尾的文件，则说明在上一次的 `swap` 过程中 Kafka 失败了，需要执行恢复操作。针对上面的情况，先去掉结尾的 `.swap`，然后判断是以 `.log` 还是以 `.index` 结尾。如果是索引文件（以 `.index` 结尾），则直接删除；如果是日志数据文件（以 `.log` 结尾），那么先删除对应的索引

文件,然后将.swap去掉表示修复成功。d.进行第二次遍历。对于目录下的每个文件,如果是索引文件,则寻找对应的.log文件,如果不存在则抛出告警信息并直接处理该索引文件,如果存在则不作任何处理;但如果该文件本身就是日志数据文件,则必然是000000...0000【offset】.log这样的形式。e.提取基础offset,并判断是否存在对应的索引文件,然后创建新的日志段对象。f.创建日志段之后判断是否存在索引文件,如果不存在则重建索引。g.将新创建的日志段加入日志段Map中,至此,第二次遍历完成。h.此时判断日志段Map中是否存在任何日志段,如果不存在则创建一个offset为0的空日志段(因为每个日志至少要有有一个日志段);如果Map中的确有日志段,则先调用recoverLog方法(稍后会介绍)恢复日志段,然后重设activeSegment的索引长度(否则容易引发日志段切分)。j.最后为每个日志段检查对应的索引文件(确保索引文件为空,以及索引长度为8的倍数,因为索引长度总是位移的整数倍)。

(31) recoverLog: 主要为日志段Map中自恢复点起的每个日志段重建索引文件,并且删除那些位于日志和索引尾部的无效字节。如果发现确实存在无效字节,那么就把那些日志段全部删除。

(32) append: 添加给定的消息集合到当前激活的日志段中,如果满足条件则进行切分。

③ LogSegment

Segment是一个逻辑概念,为了防止Log文件过大,需将Log分成许多的LogSegments。Segment又分为两部分:MessageSet文件和Index文件,分别命名为[base_offset].log和[base_offset].index,base_offset就是该Segment的起始offset,比前一个Segment里面的offset都要大。MessageSet文件是由FileMessageSet类实现的,Index文件是由OffsetIndex类实现的。同时Index文件是可以根据MessageSet文件重建的。另外,Segment提供对于MessageSet文件的写接口,需要间隔地更新Index文件。为了尽量减小Index文件的大小,只有当写入数据大于indexIntervalBytes时,才增加一条索引读。由于user传入的是逻辑offset,因此,需要先转换为物理地址,才能从文件中读取数据(至于如何转换,请参考下面的介绍)。

其对应的代码文件是LogSegment.scala,该.scala文件有一个非线程安全的类LogSegment,用于表示日志段。该类的构造函数有6个参数,分别如下。

- (1) log: FileMessageSet定义的消息集合。
- (2) index: OffsetIndex定义的位移索引,包含位移到物理文件位置的映射。
- (3) baseOffset: 日志文件的基础位移,也就是这个日志段中最低的位移。
- (4) indexIntervalBytes: 索引文件中索引项的间隔,即Kafka查找下一个物理位置时进行线性查找的最大字节数。
- (5) rollJitterMs: 指定日志段切分时的jitter time,避免日志切分时出现惊群^①现象。
- (6) time: 一个时间变量,主要提供时间方面的服务。

① 一个事件被触发后,等候这个事件的所有线程/进程都被唤醒,但只有一个完成响应。

下面对 LogSegment 的一些关键代码进行分析。

(1) created 变量：创建一个日志段的时间信息是很有用的，所以需要有一个变量保存时间信息。

(2) size 方法：既然是保存消息的日志段，也必然有一个方法保存当前日志段占用的字节数，具体实现方法就是调用 LogSegment 包含的日志对象的 size 方法。

(3) bytesSinceLastIndexEntry 变量：这个变量主要用于判断在追加写日志的同时是否需要增加一条索引项。由于 log.index.interval.bytes 默认是 4KB，因此，每写 4KB 就会在索引文件中增加一条索引记录。增加索引项之后，需要将该变量置为 0 重新计算。

(4) lastModified 及 lastModified_方法：Kafka 在清理日志段的时候，根据当前时间与该方法的返回值进行比较，清理那些陈旧的日志段，并且根据 UAP 原则提供了同名的 setter 方法用于更新日志段对象中日志文件和索引文件的最近修改时间。

(5) delete 方法：逻辑很简单的方法，就是删除日志文件和索引文件。

(6) close 方法：关闭日志段的方法，具体就是关闭底层的日志文件和索引文件。

(7) changeFileSuffixes 方法：同时更改日志文件和索引文件的扩展名。例如，在删除日志段的时候，把 a.log 和 a.index 分别更名为 a.log.delete 和 a.index.delete。

(8) flush 方法：将 Buffer 中的消息和索引项写入磁盘。

(9) nextOffset 方法：计算这个日志段中下一条消息的位移。这个方法运行起来需要很高的代价，因为它需要从索引文件中最后一项标识的位移处开始读出一个消息集合。需要特别注意的是，如果索引文件为空，则需要将整个日志段的数据都读出来，并返回一个 FetchDataInfo 对象。这个对象由一个位移元数据加上一个消息集合组成。如果这个 FetchDataInfo 对象为空，或者它包含的消息集合为空，则只返回 baseOffset。

(10) truncateTo 方法：给定一个位移，将位于该位移之后的所有索引项和日志项全部清除，如果给定的位移大于日志段本身的最大位移，则什么都不做。最后返回日志数据总共截断的字节数。值得注意的是，如果把所有日志数据都截断了，那么需要更新这个日志段的创建日期。同时还会将检查是否增加索引项的指针清零。

(11) append 方法：将一组消息追加写入以给定 offset 开始的日志段中。如果写入超过 4KB（默认的 log.index.interval.bytes 属性值），则额外写入一条新的索引项记录到索引文件中。这个方法不是线程安全的，所以在调用的时候需要有锁同步机制的保护。

(12) translateOffset 方法：给定一个 offset，找出该日志段中不小于该 offset 的第一条消息对应的物理文件位置。这个方法还有一个参数可以用来调优，不必从查询到的索引项中包含的位置开始查找，可以直接从给定的文件位置开始查找。当然，这样做的前提是必须知道这是文件中一个合法的开始位置，并且比最靠近的索引项中包含的位置值要大。

(13) read 方法：给定一个 offset，从不小于这个 offset 处的第一条开始读消息，不能超过 maxSize 字节，也必须在 maxOffset（如果提供了 maxOffset）处结束，将读到的这些消息封装到一个 FetchDataInfo 对象里返回。FetchDataInfo 由一个日志位移元数据 LogOffsetMetadata 对象和一个消息

集合组成，所谓的 `LogOffsetMetadata` 就是由消息 `offset` 加上该日志段的基础位移再加上日志段内的相对物理位置组成的。这个方法有一个关键问题：要读取消息集合到底多少字节？如果给定的 `maxSize` 是 0，那么就返回一个空的消息集合；如果 `maxSize` 大于 0 且没有指定 `maxOffset`，那么就表示能够读取最多 `maxSize` 字节的消息；而如果 `maxSize` 大于 0 且指定了 `maxOffset`，那么程序就需要计算一下 `maxOffset` 所表示的物理文件位置与起始位置的差距和 `maxSize` 谁大谁小，同时也只能选取小的值作为最终的可读取字节数。

(14) `recover` 方法：恢复一个日志段，即根据日志文件重建索引并删除那些无效的字节。所谓的无效字节是由参数限定的，任何在 `maxMessageSize` 之外的字节都被视为无效状态。该方法的实现也很简单，就是先将索引项全部截断并将索引文件重置为原来的大小，然后遍历该消息集合，超过 `indexIntervalBytes` 之后就追加一条索引记录，从而达到重建索引的目的。

④ `FileMessageSet`

`FileMessageSet` 类继承自 `MessageSet` 类，通过 `FileChannel` 可以读/写文件，是 `Segment` 中实际存放 `ByteMessageSet` 消息的文件对象。`FileChannel` 是 Java NIO 提供的类，实际上写日志时用的是 `RandomAccessFile` 来打开文件，允许来回读/写文件，也可以指定位置读/写文件。`FileMessageSet` 类有一个起始和结束的指针标识消息集合的起始位置和结束位置，这样就能实现从整个消息集合中切片的功能。该类有 5 个构造函数参数。

(1) `file`：日志文件。

(2) `channel`：底层用到的文件通道 (`File Channel`)。

(3) `start/end`：消息集合在文件中的绝对起始位置/绝对结束位置。

(4) `isSlice`：是否从整个消息集合中切分出一个切片。

(5) 除了主构造函数外，还提供了很多便利的辅助构造函数。

另外，`FileMessageSet` 类定义了一个 `_size` 变量，用于保存消息集合的字节数（同时考虑了是否支持切片）。如果不是一个切片，则将底层文件通道的指针移到最后一个字节。该类定义的方法如下。

(1) `read`：从日志文件中的指定位置读取指定大小的 `Buffer` 并封装到一个 `FileMessageSet` 对象里返回。

(2) `sizeInBytes`：该文件消息集合的字节数。

(3) `searchFor`：从给定位置处开始向后寻找不小于 `targetOffset` 的位移，并返回实际的物理文件位置。如果没有找到，则直接返回 `null`。

(4) `writeTo`：写入这个消息集合到指定的 `Channel`，允许从指定的位置写入指定大小的字节数，并返回真实写入的字节数。

(5) `iterator`：获取遍历该消息集合的迭代器，只进行一层迭代。

(6) `append`：将保存在一个 `ByteBuffer` 中的一组消息追加到指定的该消息集合所在的 `Channel` 尾部，并增加总的消息集合字节数。

(7) `flush`：提交所有已写数据到物理磁盘。

(8) close: 先调用 flush 接口将数据存入磁盘, 然后关闭 Channel。

(9) delete: 从文件系统中删除消息集合。

(10) truncateTo: 将文件消息集合截断成指定的字节大小。

(11) readInto: 将底层的文件从给定的位置开始读取内容到一个 ByteBuffer 中。

(12) renameTo: 更改消息集合底层的文件名。

除了 FileMessageSet 类外, 该 Scala 还定义了一个 Object——LogFlushStats, 里面只定义了一个定时器, 用于统计写入日志段到文件的时间。

⑤ OffsetIndex

Segment 的 Index 文件, 这是在 0.8 版本之后加上的, 之前 Message 直接使用物理 offset 标识。新版本中还是改成了使用逻辑 offset, 让物理地址对用户透明, 这样就需要一个 index 来匹配逻辑 offset 和物理地址。index 考虑到效率, 最好放在内存中, 但是考虑到大小问题, 所以使用 MappedByteBuffer (参考 Java RandomAccessFile 的用法)。Index 由 entry 组成, 每个 entry 为 8 字节, 逻辑 offset 为 4 字节, 物理地址为 4 字节。Index 是稀疏的, 不保证每个 Message 在 Index 都有索引的 entry。

逻辑 offset 是基于基础 offset 的相对 offset, 否则无法保证只使用 4 字节。例如, baseoffset 为 50, 那么 offset 55 的逻辑 offset 为 5。

物理地址是该 offset 所对应的 Message 所在日志文件的位置。因为只有 4 字节, 所以每个日志文件最大只能为 4GB。

下面开始分析代码, 首先从构造函数开始。该构造函数接收 3 个变量: 一个表示索引文件的文件变量、一个基础 offset 和一个表示最大索引文件字节数的变量。该类还定义了一些类成员变量和很多方法, 下面逐一分析。

(1) lock: 私有字段, 使用 ReentrantLock 实现, 用于同步访问 MappedByteBuffer。ReentrantLock 提供与 synchronized 相同的内存和并发性语义, 而且性能也更好。

(2) roundToExactMultiple: 私有方法, 就是计算小于第一个参数的第二个参数的最大整数倍, 比如, roundToExactMultiple(67,8)返回 64。

(3) mmap: 私有字段, 负责初始化包含该索引的内存映射对象。首先检查给定的 File 对象, 如果不存在, 则预先创建出来并设定长度为 maxIndexSize, 在设定好开始的位置之后返回。

(4) size: 私有字段, 索引文件中当前保存的索引项 (每项都是 8 字节)。

(5) maxEntries: 成员变量, 索引文件所能包含的最大索引项个数。

(6) relativeOffset: 返回根据基础 offset 的第 n 个位移。假设 n 是 5, 每项是 8 字节, 那么相对位移的值 (使用 4 字节) 必然是保存在 Buffer 中的第 40~43 字节。

(7) physical: 获取第 n 个位移对应的物理文件位置 (依然是 4 字节)。假设 $n=5$, 那么返回的值就是第 44~47 字节处保存的值。

(8) readLastEntry: 读取索引文件中最后一项对应的 OffsetPosition。

(9) lastOffset: 返回索引文件中最后一个索引项的位移。

(10) maybeLock: 在一个锁保护的情况下执行给定的方法。

(11) indexSlotFor: 以二分查找的方式寻找比给定 offset 小的最大 offset。当然, 如果最小的位移都比给定的 offset 大, 或者索引文件为空, 则直接返回-1。

(12) lookup: 计算比给定 offset 小的最大位移, 找到后返回 offset 对应物理文件位置的映射对。

(13) entry: 返回索引文件中的第 n 个位移映射对。

(14) append: 在给定的 OffsetPosition 处插入一个索引项。既然称为 append, 那么该项给定的 offset 必须比现有的所有索引项都要大。

(15) isFull: 判断该索引文件是否已满。

(16) truncate: 删除所有索引项。

(17) truncateToEntries: 删除索引项到给定的数目。

(18) truncateTo: 删除那些位移不小于给定 offset 的所有索引项。

(19) resize: 重设索引文件的大小, 在新的日志 Segment 被切分的时候调用。需要注意的是, 代码中区分了操作系统, 因为 Windows 平台不允许调整内存映射文件的长度。

(20) forceUnmap: 主要在 Windows 平台上使用。因为在 Windows 平台上修改文件长度时需要先释放内存映射对象。

(21) trimToValidSize: 调整为当前索引文件的真实占用字节大小。

(22) flush: 调用 MappedByteBuffer 的 force 方法, 将对 Buffer 的修改写入底层的文件。

(23) delete: 删除该索引文件。

(24) entries: 返回索引文件中的索引项数。

(25) sizeInBytes: 索引文件当前使用的索引项字节总数。

(26) close: 调用 trimToValidSize 方法关闭索引。

(27) renameTo: 重命名索引文件。

(28) sanityCheck: 对索引文件进行完整性检查, 包括索引文件字节数是否为 8 的整数倍、当前最大位移是否小于基础位移等。

⑥ OffsetPosition

这是一个 case classe 类, 是逻辑 offset 与物理地址直接的映射关系类。

```
/**
```

```
 * The mapping between a logical log offset and the physical position
 * in some log file of the beginning of the message set entry with the
 * given offset.
 */
```

```
case class OffsetPosition(val offset: Long, val position: Int)
```

2) 其他包相关组件

① LogOffsetMetadata

```
/*
```

```
 * A log offset structure, including:
```

```

* 1. the message offset
* 2. the base message offset of the located segment
* 3. the physical position on the located segment
*/
case class LogOffsetMetadata(messageOffset: Long,
                              segmentBaseOffset: Long =
LogOffsetMetadata.UnknownSegBaseOffset,
                              relativePositionInSegment: Int =
LogOffsetMetadata.UnknownFilePosition)

```

该 Scala 文件是一组伴生对象，定义了日志位移的元数据信息。该类定义了 Kafka 的位移元数据结构，包括：

- (1) 消息位移。
- (2) 位移所在日志段的基础位移（起始位移）。
- (3) 位移所在日志段的物理位置。

该类定义了一些方法用于获取这些信息，以及使用这些信息执行一些判断操作。

(1) `messageOffsetOnly`：判断位移元数据信息是否只包括消息位移部分的数据，而其他两部分为空。

(2) `offsetOnOlderSegment`：与给定的位移元数据实例相比较，判断这个位移是否是在一个比较旧的日志段上。

(3) `offsetOnSameSegment`：与上个方法类似，只是这次比较两个位移元数据信息是否在同一个日志段上。

(4) `precedes`：比较这个位移是否在给定位移之前。

(5) `offsetDiff`：计算此位移与给定位移之间所含的消息数。

(6) `positionDiff`：计算此位移与给定位移之间所差的字节数，前提是两个位移位于同一日志段，且此位移在给定位移之前出现。实现方法就是元数据信息中的段内相对物理位置相减。

再说说 `LogOffsetMetadata` Object。它定义了 3 个常量，分别代表未知位移的元数据、未知的段起始位移和未知的段内物理文件位置。最后，该 Object 还定义了一个 `OffsetOrdering` 嵌套类，该类实现了 Scala 的 `Ordering` 接口，因而支持两个位移元数据实例的比较。`compare` 方法通过调用两个元数据的 `offsetDiff` 方法获取两个元数据之间的消息差值。

② FetchDataInfo

这是一个简单的 case 类，由一个 `LogOffsetMetadata` 和一个 `MessageSet` 组成。

```
case class FetchDataInfo(fetchOffset: LogOffsetMetadata, messageSet: MessageSet)
```

③ ByteBufferMessageSet

该类继承自 `MessageSet` 类，是实际写入 Log 文件的结构。首先来看看这几个带 `MessageSet` 后缀的类的关系。

```
ByteBufferMessageSet(val buffer: ByteBuffer) extends MessageSet with Logging
```

`ByteBufferMessageSet` 直接继承 `MessageSet`，而 `MessageSet` 继承 `scala.collection.Iterable`。

MessageSet extends Iterable[MessageAndOffset]

即 MessageSet 是 MessageAndOffset 类的集合。

case class MessageAndOffset(message: Message, offset: Long)

MessageAndOffset 是一个 case 类，带有 Message 和 offset 两个成员。

从名称就知道是带 ByteBuffer 的 MessageSet 类，其构造函数类会调用 create 函数，里面会创建一个 ByteBuffer。

```
val buffer = ByteBuffer.allocate(MessageSet.messageSetSize(messages))
for(message <- messages)
  writeMessage(buffer,message,offsetCounter.getAndIncrement)
buffer.rewind()
buffer
```

上面的 writeMessage 函数代码如下：

```
private[kafka] def writeMessage(buffer: ByteBuffer,message: Message,offset:
Long) {
  buffer.putLong(offset)
  buffer.putInt(message.size)
  buffer.put(message.buffer)
  message.buffer.rewind()
}
```

从上面的函数可以看出，在 Buffer 里先写 offset 和 message.size 再写消息，这样就可以看出不压缩时消息的存储格式。

ByteBufferMessageSet 的消息格式如下：

```
MessageSet => [Offset MessageSize Message]
Offset => int64
MessageSize => int32
```

Message 的消息格式如下：

```
Message => Crc MagicByte Attributes Key Value
Crc => int32
MagicByte => int8
Attributes => int8
Key => bytes
Value => bytes
```

④ Log.append 日志流程

流程图如图 4.29 所示。

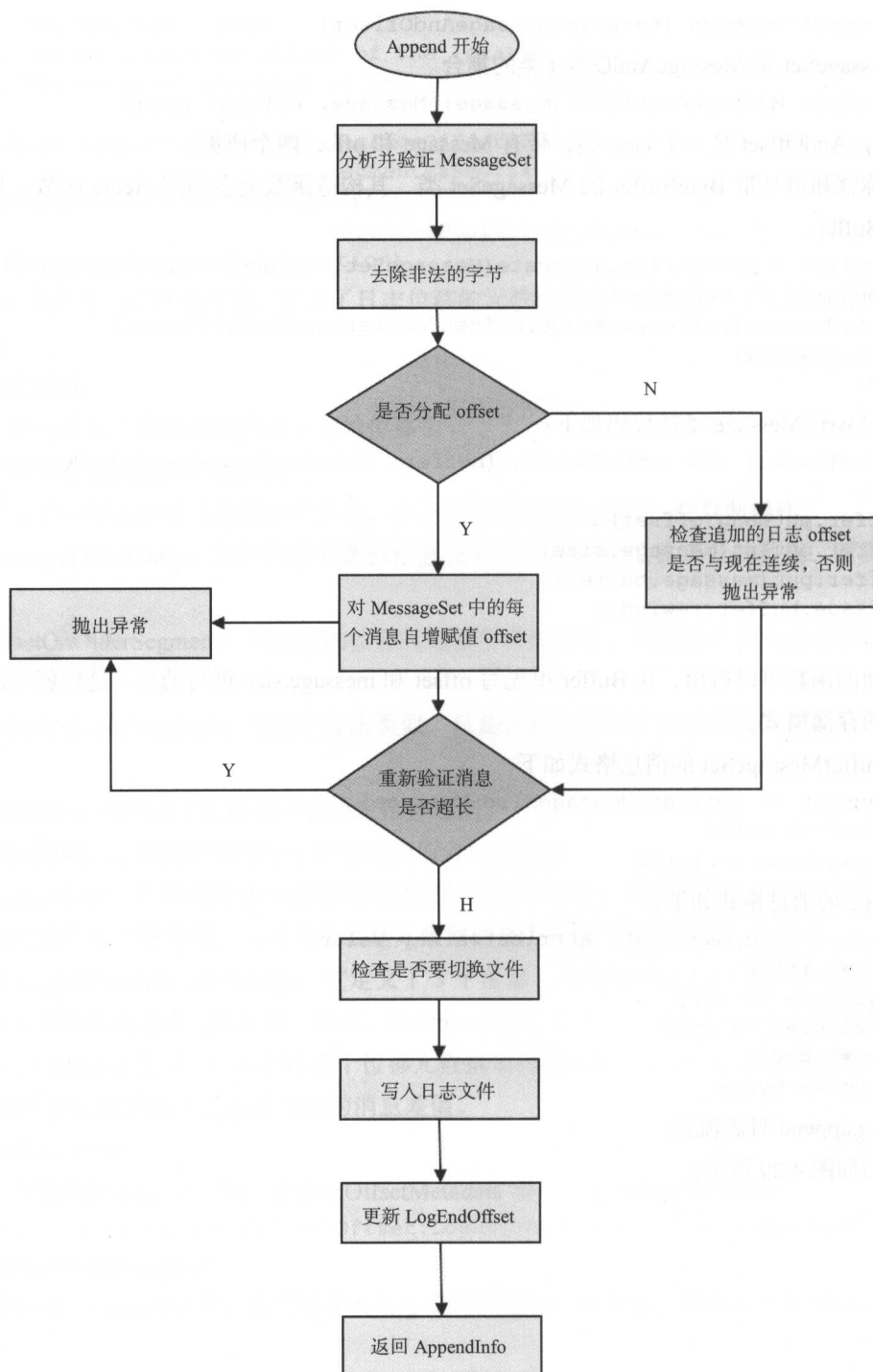


图 4.29

该函数把消息集合写入真正的日志文件中，并返回 `LogAppendInfo`。

```
def append(messages: ByteBufferMessageSet, assignOffsets: Boolean = true):
LogAppendInfo = {
    val appendInfo = analyzeAndValidateMessageSet(messages)
    // if we have any valid messages, append them to the log
    if(appendInfo.shallowCount == 0)
        return appendInfo
    //去掉一些不合法的字节，这些不合法的字节是通过检查 CRC 值得来的
    // trim any invalid bytes or partial messages before appending it to the on-disk
log
    var validMessages = trimInvalidBytes(messages, appendInfo)
    try {
        // they are valid, insert them in the log
        lock synchronized {
            // nextOffsetMetadata 是一个 LogOffsetMetadata，通过 updateLogEndOffset 函
数每次更新 messageOffset 字段，就能得到当前日志的 lastOffset，下一次写从这个 offset 计算出写入
的 offset
            appendInfo.firstOffset = nextOffsetMetadata.messageOffset
            if(assignOffsets) {
                // assign offsets to the message set
                val offset = new AtomicLong(nextOffsetMetadata.messageOffset)
                try {
                    //对 validMessages(ByteBufferMessageSet) 消息组里面的每条消息的第一个
字段 offset 赋值，这样每条写到日志里面的消息头就有 offset 了
                    validMessages = validMessages.assignOffsets(offset, appendInfo.
codec)
                } catch {
                    case e: IOException => throw new KafkaException("Error in validating
messages while appending to log '%s'".format(name), e)
                }
                // offset 在 validMessages.assignOffsets 中每遇到一条消息就会自增，所以
lastoffset 就是 offset 值减 1
                appendInfo.lastOffset = offset.get - 1
            } else {
                // we are taking the offsets we are given
                if(!appendInfo.offsetsMonotonic || appendInfo.firstOffset <
nextOffsetMetadata.messageOffset)
                    throw new IllegalArgumentException("Out of order offsets found in " +
messages)
            }
            //在 assignOffsets 里会重新压缩，需要检查消息长度是否过长
            // re-validate message sizes since after re-compression some may exceed
the limit
            for(messageAndOffset <- validMessages.shallowIterator) {
                if(MessageSet.entrySize(messageAndOffset.message) >
config.maxMessageSize) {
                    // we record the original message set size instead of trimmed size
                    // to be consistent with pre-compression bytesRejectedRate recording
```

```

        BrokerTopicStats.getBrokerTopicStats(topicAndPartition.topic).bytesRejectedRate.mark(messages.sizeInBytes)
        BrokerTopicStats.getBrokerAllTopicsStats.bytesRejectedRate.mark(messages.sizeInBytes)
        throw new MessageSizeTooLargeException("Message size is %d bytes which exceeds the maximum configured message size of %d."
            .format(MessageSet.entrySize(messageAndOffset.message), config.maxMessageSize))
    }
}
// check messages set size may be exceed config.segmentSize
if(validMessages.sizeInBytes > config.segmentSize) {
    throw new MessageSetSizeTooLargeException("Message set size is %d bytes which exceeds the maximum configured segment size of %d."
        .format(validMessages.sizeInBytes, config.segmentSize))
}
//如果当前的消息添加后超过 active segments 的文件长度，则创建一个新的日志文件再添加
// maybe roll the log if this segment is full
val segment = maybeRoll(validMessages.sizeInBytes)
//把消息追加到 active segment 中，如果字节数足够，就调用 OffsetIndex.append
添加索引
// now append to the log
segment.append(appendInfo.firstOffset, validMessages)
//更新 nextOffsetMetadata 变量到最新的 offset
// increment the log end offset
updateLogEndOffset(appendInfo.lastOffset + 1)
trace("Appended message set to log %s with first offset: %d, next offset: %d, and messages: %s"
    .format(this.name, appendInfo.firstOffset, nextOffsetMetadata.messageOffset, validMessages))
if(unflushedMessages >= config.flushInterval)
    flush()
appendInfo
}
} catch {
    case e: IOException => throw new KafkaStorageException("I/O exception in append to log '%s'".format(name), e)
}
}

```

⑤ maybeRoll

下面再来看看 maybeRoll。

```

private def maybeRoll(messagesSize: Int): LogSegment = {
    val segment = activeSegment
    if (segment.size > config.segmentSize - messagesSize ||
        segment.size > 0 && time.milliseconds - segment.created > config.segmentMs - segment.rollJitterMs ||
        segment.index.isFull) {

```

```

    debug("Rolling new log segment in %s (log_size = %d/%d,index_size =
%d/%d,age_ms = %d/%d).".
        .format(name,
            segment.size,
            config.segmentSize,
            segment.index.entries,
            segment.index.maxEntries,
            time.milliseconds - segment.created,
            config.segmentMs - segment.rollJitterMs))

    roll()
} else {
    segment
}
}

```

如果当前的消息添加后超过 active segment 的文件长度，或者 segment 创建时间太久，就会拆分文件，否则直接返回 active segment。

Roll 代码和注释如下：

```

/**
 * Roll the log over to a new active segment starting with the current
logEndOffset.
 * This will trim the index to the exact size of the number of entries it currently
contains.
 * @return The newly rolled segment
 */
def roll(): LogSegment = {
    val start = time.nanoseconds
    lock synchronized {
        val newOffset = logEndOffset
        val logFile = logFilename(dir,newOffset)
        val indexFile = indexFilename(dir,newOffset)
        //如果文件存在，则先删除
        for(file <- List(logFile,indexFile); if file.exists) {
            warn("Newly rolled segment file " + file.getName + " already exists;
deleting it first")
            file.delete()
        }
        segments.lastEntry() match {
            case null =>
            case entry => entry.getValue.index.trimToValidSize()
        }
        //生成一个新的 LogSegment
        val segment = new LogSegment(dir,
            startOffset = newOffset,
            indexIntervalBytes = config.indexInterval,
            maxIndexSize = config.maxIndexSize,
            rollJitterMs = config.randomSegmentJitter,
            time = time)

        //添加到 segments 列表中
    }
}

```

```

        val prev = addSegment(segment)
        if(prev != null)
            throw new KafkaException("Trying to roll a new log segment for topic
partition %s with start offset %d while it already exists.".format(name,newOffset))
        //调度一个异步刷盘操作
        // schedule an asynchronous flush of the old segment
        scheduler.schedule("flush-log",() => flush(newOffset),delay = 0L)
        info("Rolled new log segment for '" + name + "' in %.0f ms.".format
((System.nanoTime - start) / (1000.0*1000.0)))
        segment
    }
}

```

再来看看异步刷盘（flush）的功能。

```

/**
 * Flush log segments for all offsets up to offset-1
 * @param offset The offset to flush up to (non-inclusive); the new recovery point
 */
def flush(offset: Long) : Unit = {
    if (offset <= this.recoveryPoint)
        return
    debug("Flushing log '" + name + " up to offset " + offset + ",last flushed: " +
lastFlushTime + " current time: " +
        time.milliseconds + " unflushed = " + unflushedMessages)
    for(segment <- logSegments(this.recoveryPoint,offset))
        segment.flush()
    lock synchronized {
        if(offset > this.recoveryPoint) {
            this.recoveryPoint = offset
            lastflushedTime.set(time.milliseconds)
        }
    }
}

```

该函数主要用于读取从 recoveryPoint 到 offset 之间日志段的刷盘，而 Segment.flush 最后会分别调用 FileMessageSet 和 OffsetIndex 的 flush 函数刷盘，完成后更新 recoveryPoint 到 offset。

⑥ Segment.append

该函数是 LogSegment 提供的 append，其作用是将一组消息追加写入以给定 offset 开始的日志段中。如果写入超过 4KB（默认的 log.index.interval.bytes 属性值），则额外写入一条新的索引项记录到索引文件中。这个方法不是线程安全的，所以在调用的时候需要有锁同步机制的保护。

```

/**
 * Append the given messages starting with the given offset. Add
 * an entry to the index if needed.
 *
 * It is assumed this method is being called from within a lock.
 *
 * @param offset The first offset in the message set.

```

```

* @param messages The messages to append.
*/
@nonthreadsafe
def append(offset: Long, messages: ByteBufferMessageSet) {
    if (messages.sizeInBytes > 0) {
        trace("Inserting %d bytes at offset %d at position %d".format(messages.
sizeInBytes, offset, log.sizeInBytes()))
        // append an entry to the index (if needed)
        //如果自上次写入 Index 到现在写入 Log 的字节大于配置的 indexIntervalBytes, 则向索引
文件中写入当前 offset
        if (bytesSinceLastIndexEntry > indexIntervalBytes) {
            index.append(offset, log.sizeInBytes())
            this.bytesSinceLastIndexEntry = 0
        }
        //调用 FileMessageSet.append, 把消息写入 Channel 中
        // append the messages
        log.append(messages)
        this.bytesSinceLastIndexEntry += messages.sizeInBytes
    }
}

```

FileMessageSet.append 的代码比较简单, 直接写入 FileChannel 中。

```

/**
 * Append these messages to the message set
 */
def append(messages: ByteBufferMessageSet) {
    val written = messages.writeTo(channel, 0, messages.sizeInBytes)
    _size.getAndAdd(written)
}

```

OffsetIndex.append 的代码如下:

```

/**
 * Append an entry for the given offset/location pair to the index. This entry
must have a larger offset than all subsequent entries.
 */
def append(offset: Long, position: Int) {
    inLock(lock) {
        require(!isFull, "Attempt to append to a full index (size = " + size + ").")
        if (size.get == 0 || offset > lastOffset) {
            debug("Adding index entry %d => %d to %s.".format(offset, position,
file.getName))
            this.mmap.putInt((offset - baseOffset).toInt)
            this.mmap.putInt(position)
            this.size.incrementAndGet()
            this.lastOffset = offset
            require(entries * 8 == mmap.position, entries + " entries but file position
in index is " + mmap.position + ".")
        } else {
            throw new InvalidOffsetException("Attempt to append an offset (%d) to
position %d no larger than the last offset appended (%d) to %s."

```

```

        .format(offset, entries, lastOffset, file.getAbsolutePath())
    }
}

```

代码里面就比较明显了，按照前面章节 OffsetIndex 的写入描述，先写 4 字节的 offset-baseOffset，然后再写入日志在 Log 文件中的位置。

⑦ Log.read 日志流程

流程图如图 4.30 所示。

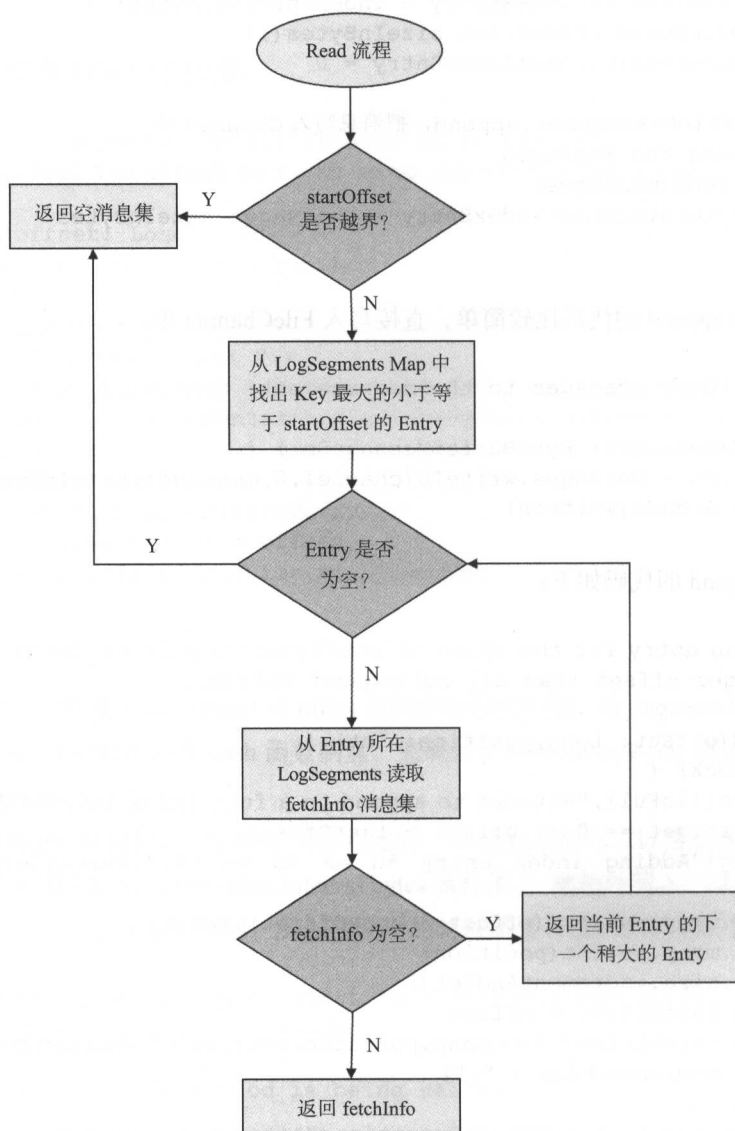


图 4.30

该函数通过指定开始读的 `startOffset` 和最大读长度等参数, 返回 `FetchDataInfo` 信息。原理很简单, 就是从保存的 `segments map` 中找到 `baseOffset` 与 `startOffset` 最接近的 `segment`, 开始查找和读取数据。

```
/**
 * Read messages from the log
 *
 * @param startOffset The offset to begin reading at
 * @param maxLength The maximum number of bytes to read
 * @param maxOffset -The offset to read up to, exclusive. (i.e. the first offset
 NOT included in the resulting message set).
 *
 * @throws OffsetOutOfRangeException If startOffset is beyond the log end offset
 or before the base offset of the first segment.
 * @return The fetch data information including fetch starting offset metadata
 and messages read
 */
def read(startOffset: Long, maxLength: Int, maxOffset: Option[Long] = None):
FetchDataInfo = {
    trace("Reading %d bytes from offset %d in log %s of length %d bytes".format
(maxLength, startOffset, name, size))
    // check if the offset is valid and in range
    //检查一下 startOffset 是否是 nextOffset, 如果是则表明日志还不存在, 返回空消息集
    val next = nextOffsetMetadata.messageOffset
    if(startOffset == next)
        return FetchDataInfo(nextOffsetMetadata, MessageSet.Empty)
    //返回一个 Map.Entry(baseOffset, LogSegment), 其 baseOffset 永远小于等于 startOffset
    var entry = segments.floorEntry(startOffset)
    // attempt to read beyond the log end offset is an error
    if(startOffset > next || entry == null)
        throw new OffsetOutOfRangeException("Request for offset %d but we only have
log segments in the range %d to %d.".format(startOffset, segments.firstKey, next))
    // do the read on the segment with a base offset less than the target offset
    // but if that segment doesn't contain any messages with an offset greater
than that
    // continue to read from successive segments until we get some messages or
we reach the end of the log
    while(entry != null) {
        //调用 LogSegment.read 获得 fetchInfo
        val fetchInfo = entry.getValue.read(startOffset, maxOffset, maxLength)
        //如果为空, 则寻找其连续的下一个 LogSegment 去读取
        if(fetchInfo == null) {
            entry = segments.higherEntry(entry.getKey)
        } else {
            return fetchInfo
        }
    }
}
// okay we are beyond the end of the last segment with no data fetched although
```

```

the start offset is in range,
    // this can happen when all messages with offset larger than start offsets
have been deleted.
    // In this case, we will return the empty set with log end offset metadata
    FetchDataInfo(nextOffsetMetadata, MessageSet.Empty)
}

```

⑧ LogSegment.read

```

/**
 * Read a message set from this segment beginning with the first offset >=
startOffset. The message set will include
 * no more than maxSize bytes and will end before maxOffset if a maxOffset is
specified.
 *
 * @param startOffset A lower bound on the first offset to include in the message
set we read
 * @param maxSize The maximum number of bytes to include in the message set
we read
 * @param maxOffset An optional maximum offset for the message set we read
 *
 * @return The fetched data and the offset metadata of the first message whose
offset is >= startOffset,
 *         or null if the startOffset is larger than the largest offset in
this log
 */
@threadsafe
def read(startOffset: Long, maxOffset: Option[Long], maxSize: Int): FetchDataInfo
= {
    if (maxSize < 0)
        throw new IllegalArgumentException("Invalid max size for log read
(%d)".format(maxSize))
    //获得该 LogSegment 的 size
    val logSize = log.sizeInBytes // this may change, need to save a consistent
copy
    //把逻辑 offset 转换为实际物理地址, 并返回一个 OffsetPosition, 其 offset 是大于等于
startOffset 的。后面会对这个函数进行详细介绍
    val startPosition = translateOffset(startOffset)
    // if the start position is already off the end of the log, return null
    if (startPosition == null)
        return null
    //生成一个 LogOffsetMetadata 对象, 包含逻辑 offset 和物理地址等信息
    val offsetMetadata = new LogOffsetMetadata(startOffset, this.baseOffset,
startPosition.position)
    // if the size is zero, still return a log segment but with zero size
    if (maxSize == 0)
        return FetchDataInfo(offsetMetadata, MessageSet.Empty)
    //因为这个接口可以指定最大读取到的 maxOffset, 所以下面会计算实际最大能读取的长度
    // calculate the length of the message set to read based on whether or not
they gave us a maxOffset

```

```

val length =
    maxOffset match {
        case None =>
            // no max offset, just use the max size they gave unmolested
            maxSize
        case Some(offset) => {
            // there is a max offset, translate it to a file position and use that
            to calculate the max read size
            if (offset < startOffset)
                throw new IllegalArgumentException("Attempt to read with a maximum
            offset (%d) less than the start offset (%d)".format(offset, startOffset))
            //再次把 max offset 转换为物理地址, 然后取 maxSize 与 maxOffset 到 startOffset
            之间的最小值作为读取长度
            val mapping = translateOffset(offset, startPosition.position)
            val endPosition =
                if (mapping == null)
                    logSize // the max offset is off the end of the log, use the end
                of the file
            else
                mapping.position
            min(endPosition - startPosition.position, maxSize)
        }
    }
    //返回 FetchDataInfo 对象
    FetchDataInfo(offsetMetadata, log.read(startPosition.position, length))
}

```

代码中的一个主要函数是 `translateOffset`, 用于把逻辑地址转换为实际物理地址。

```

/**
 * Find the physical file position for the first message with offset >= the
 * requested offset.
 *
 * The lowerBound argument is an optimization that can be used if we already
 * know a valid starting position
 * in the file higher than the greatest-lower-bound from the index.
 *
 * @param offset The offset we want to translate
 * @param startingFilePosition A lower bound on the file position from which
 * to begin the search. This is purely an optimization and
 * when omitted, the search will begin at the position in the offset index.
 *
 * @return The position in the log storing the message with the least offset
 * >= the requested offset or null if no message meets this criteria.
 */
@threadsafe
private[log] def translateOffset(offset: Long, startingFilePosition: Int = 0):
OffsetPosition = {
    //在 index buffer 中查找最大的小于等于 offset 的 OffsetPosition
    val mapping = index.lookup(offset)
}

```

//根据 OffsetPosition 得到的物理地址，在 Log 中查找更接近的 OffsetPosition (大于等于目标 offset 的位置)

```
log.searchFor(offset,max(mapping.position,startingFilePosition))
}
```

Index.lookup 代码如下：

```
def lookup(targetOffset: Long): OffsetPosition = {
  maybeLock(lock) {
    //把 ByteBuffer 复制一份，复制的副本和 mmap 之间是共享内容的，新缓冲区的 position、
    limit、mark 和 capacity 都初始化为原始缓冲区的索引值，但是这些值是相互独立的
    val idx = mmap.duplicate
    //用二分法在 index buffer 中找到最大的小于等于 targetOffset 的位置
    val slot = indexSlotFor(idx,targetOffset)
    //找到索引所在的位置后，返回 OffsetPosistion 类
    if(slot == -1)
      OffsetPosition(baseOffset,0)
    else
      OffsetPosition(baseOffset + relativeOffset(idx,slot),physical(idx,slot))
  }
}
```

其中，relativeOffset 与 physical 函数就是根据 ByteBuffer 中的位置 slot，返回相对逻辑 offset 与物理地址的。

```
/* return the nth offset relative to the base offset */
private def relativeOffset(buffer:ByteBuffer,n:Int):Int = buffer.getInt(n * 8)
/* return the nth physical position */
private def physical(buffer: ByteBuffer,n:Int):Int = buffer.getInt(n * 8 + 4)
```

再看 FileMessageSet.searchFor，其主要做的事情是从 startingPosition 开始，按顺序找到第一个 offset 大于等于目标 offset 的消息位置，并返回 OffsetPosition 类。

```
/**
 * Search forward for the file position of the last offset that is greater than
 * or equal to the target offset
 * and return its physical position. If no such offsets are found,return null.
 * @param targetOffset The offset to search for.
 * @param startingPosition The starting position in the file to begin
 * searching from.
 */
def searchFor(targetOffset: Long,startingPosition: Int): OffsetPosition = {
  var position = startingPosition
  //分配一个 LogOverhead 大小的 buffer，用于读取消息头，LogOverhead 大小为 12
  byte(MessageSize 4,OffsetLength 8)
  val buffer = ByteBuffer.allocate(MessageSet.LogOverhead)
  val size = sizeInBytes()
  while(position + MessageSet.LogOverhead < size) {
    buffer.rewind()
    channel.read(buffer,position)
    //如果消息头都读不出来，则抛出异常
    if(buffer.hasRemaining)
```

```

        throw new IllegalStateException("Failed to read complete buffer for
targetOffset %d startPosition %d in %s"
                                .format(targetOffset, startingPosition,
file.getAbsolutePath())
        buffer.rewind()
        val offset = buffer.getLong()
        //返回大于等于 targetOffset 的 OffsetPosition
        if(offset >= targetOffset)
            return OffsetPosition(offset, position)
        val messageSize = buffer.getInt()
        if(messageSize < Message.MessageOverhead)
            throw new IllegalStateException("Invalid message size:" + messageSize)
        //position 移位到下一个消息, 具体消息布局请看前面章节
        position += MessageSet.LogOverhead + messageSize
    }
    null
}

```

至此, Log 模块的基本读/写函数已经分析完毕。

7. 生态系统^①

从 Kafka 官方网站可以看到它的生态范围非常广, 覆盖从发行版、流处理对接、Hadoop 集成、搜索集成到周边组件, 如管理、日志、发布、打包、AWS 集成等。

详细信息可以访问 <https://cwiki.apache.org/confluence/display/KAFKA/Ecosystem>。

8. Kafka 应用案例

图 4.31 展示的是一个 Kafka 应用电信场景, 生产数据中心和离线数据中心之间的数据同步。

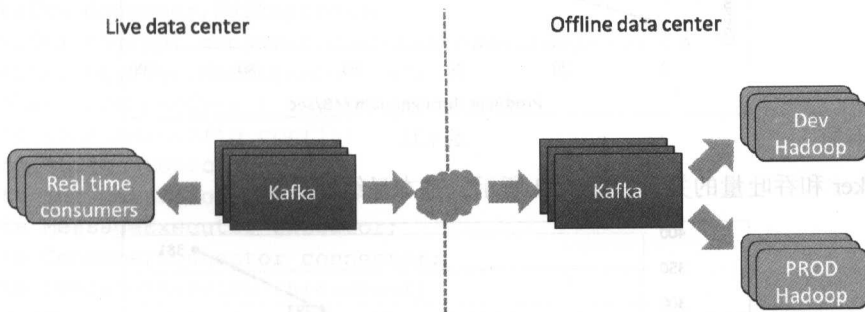


图 4.31

9. Kafka 性能

测试条件:

- 2 Linux boxes
- 16 2.0 GHz cores

^① Kafka Ecosystem, <https://cwiki.apache.org/confluence/display/KAFKA/Ecosystem>。

- 6 7200 rpm SATA drive RAID 10
- 24GB memory
- 1Gbit/s network link
- 200 byte messages
- Producer batch size 200 messages
- Producer batch size = 40K
- Consumer batch size = 1MB
- 100 topics, broker flush interval = 100K
 - Producer throughput = 90 MB/sec
 - Consumer throughput = 60 MB/sec
 - Consumer latency = 220 ms
- (100 topics, 1 producer, 1 broker)

测试结果如下：

(1) 吞吐量和时延的关系如图 4.32 所示。

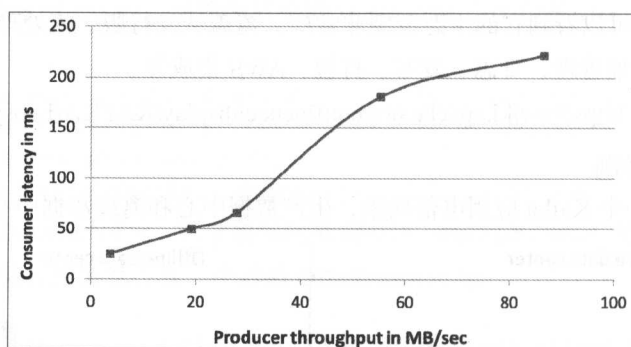


图 4.32

(2) Broker 和吞吐量的关系如图 4.33 所示，基本呈线性扩展。

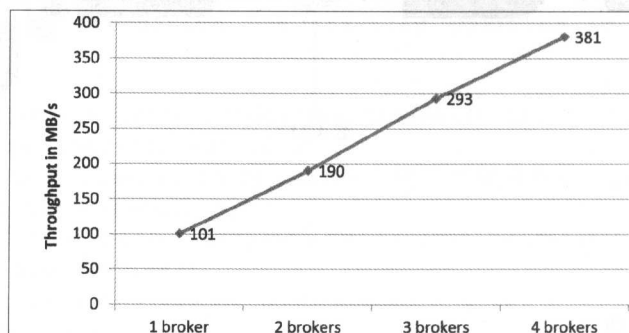


图 4.33

(3) 吞吐量和未消费数据的关系如图 4.34 所示。

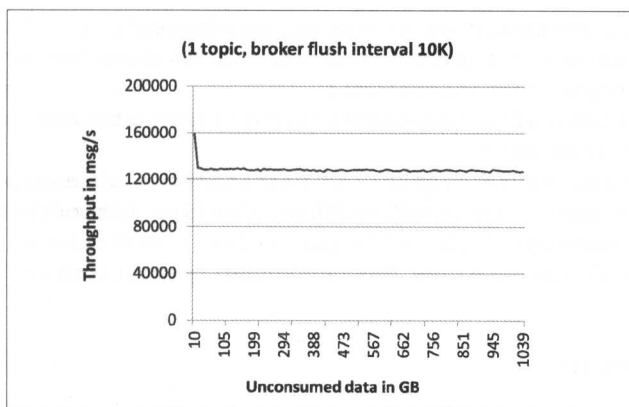


图 4.34

10. 代码样例

1) Consumer

```
package com.test.kafka;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import kafka.consumer.Consumer;
import kafka.consumer.ConsumerConfig;
import kafka.consumer.ConsumerIterator;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;
import kafka.message.MessageAndMetadata;
public class LogConsumer {
    private ConsumerConfig config;
    private String topic;
    private int partitionsNum;
    private MessageExecutor executor;
    private ConsumerConnector connector;
    private ExecutorService threadPool;
    public LogConsumer(String topic,int partitionsNum,MessageExecutor executor)
throws Exception{
        Properties properties = new Properties();

        properties.load(ClassLoader.getResourceAsStream("consumer. properties"));
        config = new ConsumerConfig(properties);
        this.topic = topic;
        this.partitionsNum = partitionsNum;
        this.executor = executor;
    }
    public void start() throws Exception{
```



```

        connector = Consumer.createJavaConsumerConnector(config);
        Map<String,Integer> topics = new HashMap<String,Integer>();
        topics.put(topic,partitionsNum);
        Map<String,List<KafkaStream<byte[],byte[]>>> streams = connector.
createMessageStreams(topics);
        List<KafkaStream<byte[],byte[]>> partitions = streams.get(topic);
        threadPool = Executors.newFixedThreadPool(partitionsNum);
        for(KafkaStream<byte[],byte[]> partition : partitions){
            threadPool.execute(new MessageRunner(partition));
        }
    }
    public void close(){
        try{
            threadPool.shutdownNow();
        }catch(Exception e){
            //
        }finally{
            connector.shutdown();
        }
    }
}
class MessageRunner implements Runnable{
    private KafkaStream<byte[],byte[]> partition;
    MessageRunner(KafkaStream<byte[],byte[]> partition) {
        this.partition = partition;
    }
    public void run(){
        ConsumerIterator<byte[],byte[]> it = partition.iterator();
        while(it.hasNext()){
            MessageAndMetadata<byte[],byte[]> item = it.next();
            System.out.println("partiton:" + item.partition());
            System.out.println("offset:" + item.offset());
            executor.execute(new String(item.message()));//UTF-8
        }
    }
}
}
interface MessageExecutor {
    public void execute(String message);
}
/**
 * @param args
 */
public static void main(String[] args) {
    LogConsumer consumer = null;
    try{
        MessageExecutor executor = new MessageExecutor() {
            public void execute(String message) {
                System.out.println(message);
            }
        };
    };
}

```

```

        consumer = new LogConsumer("test-topic",2,executor);
        consumer.start();
    }catch(Exception e){
        e.printStackTrace();
    }finally{
        //        if(consumer != null){
        //            consumer.close();
        //        }
    }
}

```

2) Producer

```

package com.test.kafka;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.Properties;
import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;
public class LogProducer {
    private Producer<String,String> inner;
    public LogProducer() throws Exception{
        Properties properties = new Properties();
        properties.load(ClassLoader.getResourceAsStream("producer.properties"));
        ProducerConfig config = new ProducerConfig(properties);
        inner = new Producer<String,String>(config);
    }
    public void send(String topicName,String message) {
        if(topicName == null || message == null){
            return;
        }
        KeyedMessage<String,String> km = new KeyedMessage<String,String> (topicName,
        message);
        inner.send(km);
    }
    public void send(String topicName,Collection<String> messages) {
        if(topicName == null || messages == null){
            return;
        }
        if(messages.isEmpty()){
            return;
        }
        List<KeyedMessage<String,String>> kms = new ArrayList<KeyedMessage<String,
        String>>();
        for(String entry : messages){
            KeyedMessage<String,String> km = new KeyedMessage<String,String>
            (topicName,entry);

```

```

        kms.add(km);
    }
    inner.send(kms);
}
public void close(){
    inner.close();
}
/**
 * @param args
 */
public static void main(String[] args) {
    LogProducer producer = null;
    try{
        producer = new LogProducer();
        int i=0;
        while(true){
            producer.send("test-topic","this is a sample" + i);
            i++;
            Thread.sleep(2000);
        }
    }catch(Exception e){
        e.printStackTrace();
    }finally{
        if(producer != null){
            producer.close();
        }
    }
}
}

```

4.7 小结

要获取大数据，首先要考虑的是数据怎么获取，这是整个大数据系统中的第一道关口，也是基础。

这些组件在部署上通常靠近数据源，所以对这些系统的小型化、性能要求会很高。另外，随着未来数据逐渐增多，对实时性的要求也越来越高。为了匹配业务的扩展，这些组件未来在实时、智能、可靠性上的要求也将越来越高。

第 5 章

流处理

我们将大数据处理按处理时间的跨度要求分为以下几类，从短到长分别是：

- 基于实时数据流的数据处理（Streaming Data Processing），通常的时间跨度在数百毫秒到数秒之间。
- 基于历史数据的交互式查询（Interactive Query），通常的时间跨度在数十秒到数分钟之间。
- 复杂的批量数据处理（Batch Data Processing），通常的时间跨度在几分钟到数小时之间。

接下来的几章会分别讲述在这几种处理时间跨度要求下将采取的技术，首先讲述实时数据流的处理。

当然，批和流及交互式查询并不一定能完全分开，Spark 的一个主要想法就是统一几个引擎，所以本章会讲到 Spark Streaming，在第 7 章将详细讲述 Spark 对批的处理。

5.1 算子

在介绍流、交互式查询、准实时、后分析之前，先简单探讨一个概念——算子，英文叫 Operation。

算子在数学上可解释为一个函数空间到函数空间上的映射 $O: X \rightarrow X$ ，这个比较拗口。对大数据处理框架来说，可以理解为一个基本处理单元，即通常映射到框架的一个函数。使用算子需要指定输入和输出，算子负责完成对应的数据的转化。一段完整的处理逻辑就是 $[1, n]$ 个算子或串行、或并行的处理逻辑叠加到一起，完成一段数据的处理。

常见的算子有 Group、Sort、Orderby、Window 等，在后面的章节里我们会不停地看到算子这个概念。

5.2 流的概念

Streaming（流）是一种数据传送技术，它把客户机收到的数据变成一个稳定连续的流，源源不断地送出。正是由于数据的传输呈持续不停的形态，所以流引擎需要持续不断地处理数据。相反，Batch 是处理完一批数据再导入下一批数据。最能体现流的特点的是 Window 算子，这也是流有别于 Batch 的最基本的算子。

以 Spark Streaming 为例，WindowOperations 有点类似于 Storm 中的 State，可以设置窗口的大小和滑动窗口的间隔来动态获取当前 Streaming 的允许状态，如图 5.1 所示。

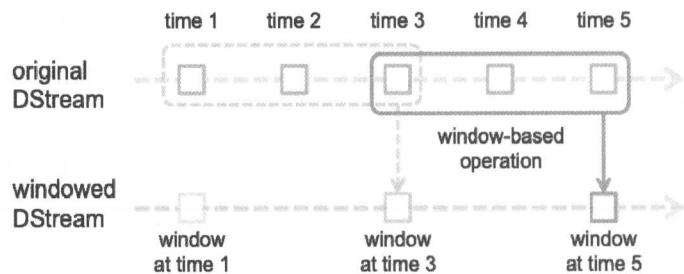


图 5.1

实际上，在实现过程中，根据实时性的要求不同，流也有两种处理方式：一种以 IBM 的 IBM InfoSphere Streams 为代表，一个消息传递过来，立刻处理；另一种是 Spark Streaming，因为 Spark 本身把数据存放在内存中处理，速度较快，所以处理流的时候以较小的批（Mini-Batch）来模拟流处理，可以轻松应对对实时性要求并不是很高的场景。

5.3 流的应用场景

5.3.1 金融领域

流应用历史最悠久且最广泛的应该是金融领域，可以通过 K 线图实时分析股票价格来判断股票是否可以买入，如图 5.2 所示。

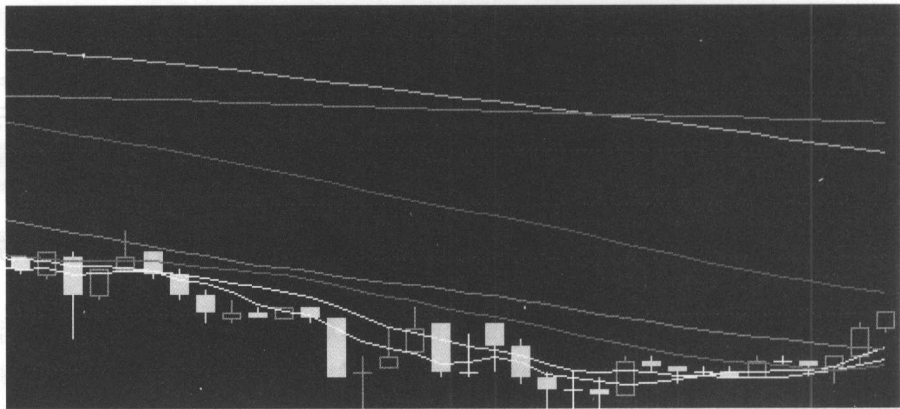


图 5.2

股票市场判断什么时候买入，一个流的规则如下：

$MA(5) < (MA(15) < MA(30) < MA(60))$ (多头排列)

5 分钟内，顺序出现：

当前价格突破 $MA(5)$

当前价格突破 $MA(15)$

① MA：表示不同时间周期的股票平均价格。文中的 $MA(5)$ 表示 5 分钟平均价格，其他依次类推。

当前价格突破 MA(30)
买入

在上面的例子中，流引擎实时计算股价，只要满足判断条件，即可买入。

5.3.2 电信领域

电信领域一个常见的样例是质差补偿。客户在通话过程中发生断线、掉话^①，需要识别出来，并通过发送道歉短信、返还话费等补偿措施来提升用户满意度，提升运营商品牌。掉话需判断是网络设备引起的，还是客户自己挂断的，这就需要采集大量设备的数据，然后统一放到流引擎中进行分析，实时判断是否采取质差补偿措施。

5.4 业界两种典型的流引擎

5.4.1 Storm^②

Storm 是 Twitter 开源的一个分布式实时数据处理系统。按照 Storm 作者的说法，Storm 对于实时计算的意义类似于 Hadoop 对于批处理的意义。我们都知道，根据 Google MapReduce 来实现的 Hadoop 为我们提供了 Map、Reduce 原语，使批处理程序变得非常简单和优美。同样，Storm 也为实时计算提供了一些简单、优美的原语。

1. Storm 的基本概念

首先通过 Storm 和 Hadoop 的对比来了解 Storm 中的基本概念，如表 5.1 所示。

表 5.1		
	Hadoop	Storm
系统角色	JobTracker	Nimbus
	TaskTracker	Supervisor
	Child	Worker
应用名称	Job	Topology
组件接口	Mapper/Reducer	Spout/Bolt

接下来具体看一下这些概念。

- Nimbus：负责资源分配和任务调度。
- Supervisor：负责接收 Nimbus 分配的任务，启动和停止属于自己管理的 Worker 进程。
- Worker：运行具体处理组件逻辑的进程。
- Task：Worker 中每一个 Spout/Bolt 的线程称为一个 Task。在 Storm 0.8 之后，Task 不再与物理线程对应，同一个 Spout/Bolt 的 Task 可能会共享一个物理线程，该线程称为 Executor。

① 掉话：通话过程中通话中断。
② Storm 简介：<http://www.searchtb.com/2012/09/introduction-to-storm.html>。

图 5.3 描述了以上几个角色之间的关系。

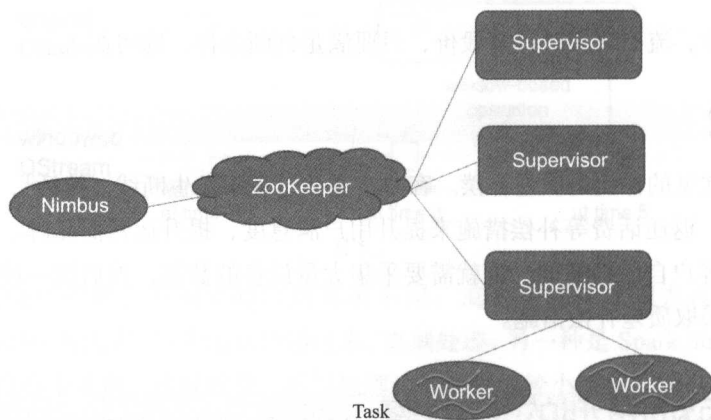


图 5.3

- **Topology**: Storm 中运行的一个实时应用程序，因为各个组件间的消息流动形成逻辑上的一个拓扑结构。
- **Spout**: 在一个 Topology 中产生源数据流的组件。通常情况下 Spout 会从外部数据源中读取数据，然后转换为 Topology 内部的源数据。Spout 是一个主动的角色，其接口中有一个 `nextTuple()` 函数，Storm 框架会不停地调用此函数，用户只需在其中生成源数据即可。
- **Bolt**: 在一个 Topology 中接收数据然后执行处理的组件。Bolt 可以执行过滤、函数操作、合并、写数据库等操作。Bolt 是一个被动的角色，其接口中有一个 `execute(Tuple input)` 函数，在接收到消息后会调用此函数，用户可以在其中执行自己想要的操作。
- **Tuple**: 一次消息传递的基本单元。本来应该是一个 Key-Value 的 Map，但是由于各个组件间传递的 Tuple 的字段名称已经事先定义好，所以在 Tuple 中只要按序填入各个 Value 就可以了，所以就是一个 Value List。
- **Stream**: 源源不断传递的 Tuple 就组成了 Stream。
- **Stream Grouping**: 即消息的 Partition 方法。Storm 中提供若干种实用的 Grouping 方式，包括 Shuffle、Fields Hash、All、Global、None、Direct、localOrShuffle 等。

相比于 S4、Puma 等其他实时计算系统，Storm 最大的亮点在于其记录级容错和能够保证消息精确处理的事务功能。下面就来重点看一下这两个亮点的实现原理。

2. Storm 记录级容错的基本原理

首先来看一下什么叫作记录级容错。Storm 允许用户在 Spout 中发射一个新的源 Tuple 时为其指定一个 message ID，这个 message ID 可以是任意的 Object 对象。多个源 Tuple 可以共用一个 message ID，表示这些源 Tuple 对用户来说是同一个消息单元。Storm 中记录级容错的意思是，Storm 会告知用户每个消息单元是否在指定时间内被完全处理了。那什么叫作完全处理呢？就是该 message ID 绑定的源 Tuple 及由该源 Tuple 后续生成的 Tuple 经过了 Topology 中每一个应该到达

的 Bolt 的处理。举个例子，在图 5.4 中，在 Spout 中由 message 1 绑定的 Tuple 1 和 Tuple 2 经过 Bolt 1 和 Bolt 2 的处理生成两个新的 Tuple，最终都流向 Bolt 3。当这个过程完成处理时，就称 message 1 被完全处理了。

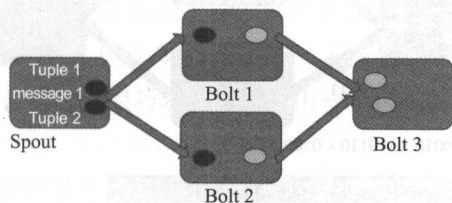


图 5.4

在 Storm 的 Topology 中有一个系统级组件，叫作 Acker。这个 Acker 的任务就是追踪从 Spout 中流出来的每个 message ID 绑定的若干 Tuple 的处理路径。如果在用户设置的最大超时时间内这些 Tuple 没有被完全处理，那么 Acker 就会告知 Spout 该消息处理失败了；相反，则会告知 Spout 该消息处理成功了。在刚才的描述中，我们提到了“记录 Tuple 的处理路径”，如果尝试过这么做的读者可以仔细思考一下这件事的复杂程度。但是在 Storm 中却使用一种非常巧妙的方法做到了。在说明这个方法之前，我们先来复习一个数学定理。

$A \text{ xor } A = 0$ 。

$A \text{ xor } B \cdots \text{ xor } B \text{ xor } A = 0$ ，其中每个操作数出现且仅出现两次。

Storm 中使用的巧妙方法就是基于这个定理的。具体过程为：在 Spout 中，系统会为用户指定的 message ID 生成一个对应的 64 位整数，作为一个 root ID。root ID 会传递给 Acker 及后续的 Bolt 作为该消息单元的唯一标识。同时无论是 Spout 还是 Bolt，每次新生成一个 Tuple 的时候，都会赋予该 Tuple 一个 64 位的整数的 ID。Spout 发射完某个 message ID 对应的源 Tuple 之后，会告知 Acker 自己发射的 root ID 及生成的那些源 Tuple 的 ID。而 Bolt 每次接收到一个输入 Tuple 并处理完之后，也会告知 Acker 自己处理的输入 Tuple 的 ID 及新生成的那些 Tuple 的 ID。Acker 只需对这些 ID 做一个简单的异或运算，就能判断出该 root ID 对应的消息单元是否处理完成了。下面通过一个图示来说明这个过程，如图 5.5 所示。

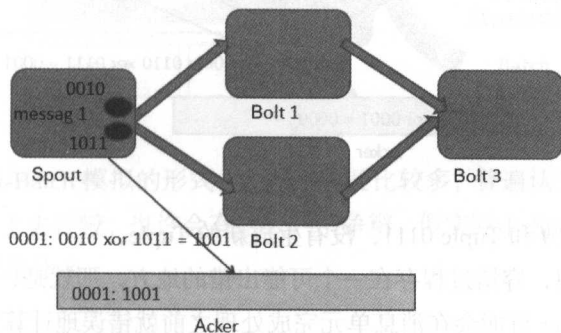


图 5.5

在 Spout 中绑定 message 1 生成了两个源 Tuple，ID 分别是 0010 和 1011，如图 5.6 所示。

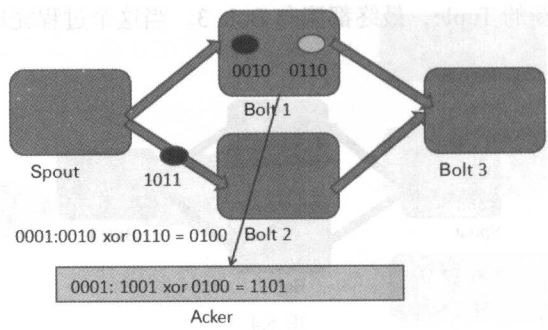


图 5.6

Bolt 1 处理 Tuple 0010 时生成了一个新的 Tuple，ID 为 0110，如图 5.7 所示。

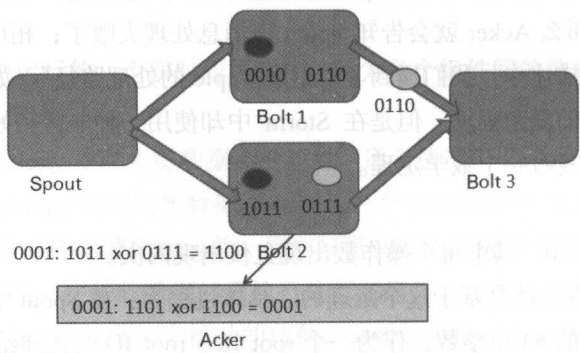


图 5.7

Bolt 2 处理 Tuple 1011 时生成了一个新的 Tuple，ID 为 0111，如图 5.8 所示。

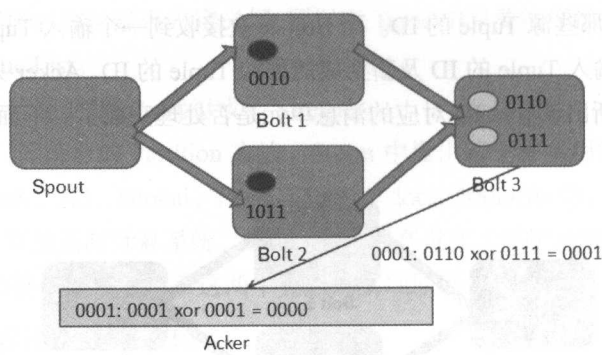


图 5.8

Bolt 3 接收到 Tuple 0110 和 Tuple 0111，没有生成新的 Tuple。

细心的读者可能会发现，容错过程存在一个可能出错的地方，那就是，如果生成的 Tuple ID 并不是完全各异的，那么 Acker 可能会在消息单元完成处理之前就错误地计算为 0。这个错误在理论上的确是存在的，但是在实际中其概率是极低的，完全可以忽略。

3. Storm 的事务拓扑

事务拓扑(Transactional Topology)是 Storm 0.7 引入的特性,在 0.8 版本中已经被封装为 Trident,提供了更加便利和直观的接口。因为篇幅所限,在此只对事务拓扑做一个简单的介绍。

事务拓扑的目的是为了满足对消息处理有着极其严格要求的场景,例如,实时计算某个用户的成交笔数,要求结果完全精确,不能多也不能少。Storm 的事务拓扑是完全基于它底层的 Spout/Bolt/Acker 原语实现的,通过一层巧妙的封装得出一个优雅的实现。笔者认为这也是 Storm 最大的魅力。

事务拓扑简单来说就是将消息分为一个个的批(Batch),同一批内的消息及批与批之间的消息可以并行处理。另外,用户可以设置某些 Bolt 为 Committer,Storm 可以保证 Committer 的 finishBatch() 操作是按严格不降序的顺序执行的。用户可以利用这个特性,通过简单的编程技巧实现精确的消息处理。

5.4.2 Spark Streaming^①

BDAS 是伯克利大数据处理协议栈,是以 Spark 为基础的一套软件栈。它利用基于内存的通用计算模型,同时支持 Batch、Interactive、Streaming 的处理,且兼容支持 HDFS 和 S3 等分布式文件系统,可以部署在 YARN 和 Mesos 等流行的集群资源管理器上。

BDAS 的架构如图 5.9 所示。其中,Spark 可以替代 MapReduce 进行批处理,利用其基于内存的特点,特别擅长迭代式和交互式数据处理;Shark 可以处理大规模数据的 SQL 查询,兼容 Hive 的 HQL。同时利用 Spark 在内存中的处理速度,以 Mini-Batch 的形式模拟 Streaming 的处理,应对对实时性要求并不是特别高的一般型应用。

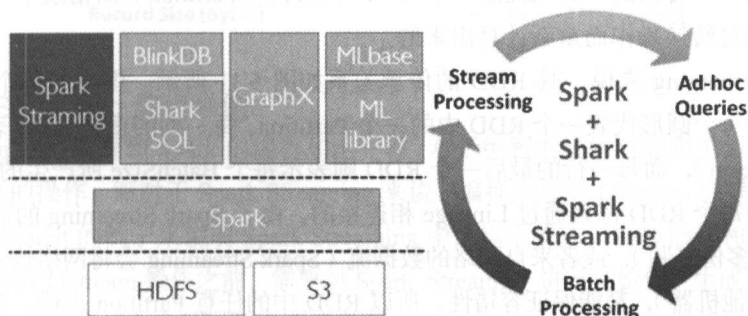


图 5.9

就 Spark 通过 Mini-Batch 模拟的形式,业界的质疑比较多,普遍认为 Spark 不是真的流系统。就这点而言,如果从学术上去计较,也许会有这么一个争辩,但实际上 Spark 是否是真的流系统并不重要,关键看适应的场景是什么。

① CSD NSpark Streaming: 大规模流式数据处理的新贵, <http://www.csdn.net/article/2014-01-28/2818282-Spark-Streaming-big-data>, <http://blog.selfup.cn/619.html>。

1. Spark Streaming 架构

计算流程：Spark Streaming 用于将流式计算分解成一系列短小的批处理作业。这里的批处理引擎是 Spark，也就是把 Spark Streaming 的输入数据按照 BatchSize（如 1 秒）分成一段一段的数据（Discretized Stream, DStream），每一段数据都转换成 Spark 中的 RDD（Resilient Distributed Dataset），然后将 Spark Streaming 中对 DStream 的 Transformation 操作转换为针对 Spark 中对 RDD 的 Transformation 操作，将 RDD 经过操作变成中间结果保存在内存中。整个流式计算根据业务的需求可以对中间结果进行叠加，或者存储到外部设备。图 5.10 展示了 Spark Streaming 的整个流程。

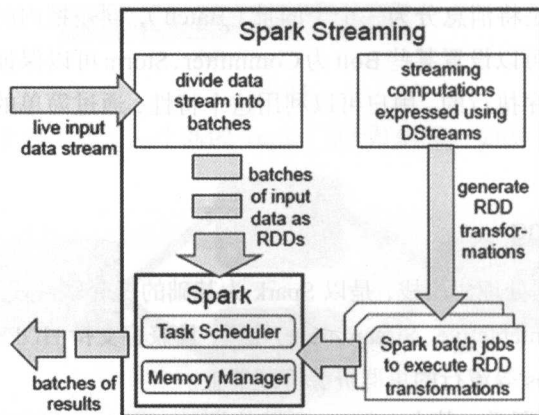


图 5.10

容错性：对于流式计算来说，容错性至关重要。首先需要明确一下 Spark 中 RDD 的容错机制。每个 RDD 都是一个不可变的分布式、可重算的数据集，其记录着确定性的操作继承关系（Lineage），所以只要输入数据是容错的，那么任意一个 RDD 的分区（Partition）出错或不可用，都是可以利用原始输入数据通过转换操作而重新计算出来的。

对于 Spark Streaming 来说，其 RDD 的传承关系如图 5.11 所示。图中的每个椭圆形表示一个 RDD，椭圆形中的每个圆形代表一个 RDD 中的一个 Partition，每一列的多个 RDD 表示一个 DStream（图中有三个 DStream），而每一行的最后一个 RDD 则表示每个 BatchSize 所产生的中间结果 RDD。可以看到，图中的每个 RDD 都是通过 Lineage 相连接的，由于 Spark Streaming 的输入数据可以来自磁盘，如 HDFS（多份复制），或者来自网络的数据流（Spark Streaming 会将网络输入数据的每一个数据流复制两份到其他机器），都能保证容错性，所以 RDD 中的任意 Partition 出错，都可以并行地在其他机器上将缺失的 Partition 计算出来。这种容错恢复方式比连续计算模型（如 Storm）的效率更高。

实时性：对于实时性的讨论，会涉及流式处理框架的应用场景。Spark Streaming 将流式计算分解成多个 SparkJob，对于每一段数据的处理都会经过 SparkDAG 图分解，以及 Spark 任务集的调度过程。对于目前版本的 Spark Streaming 而言，其最小的 BatchSize 在 0.5~2s 之间（Storm 目前最小的延迟在 100ms 左右），所以 Spark Streaming 能够满足除对实时性要求非常高（如高频实时交易）的所有流式准实时计算场景。

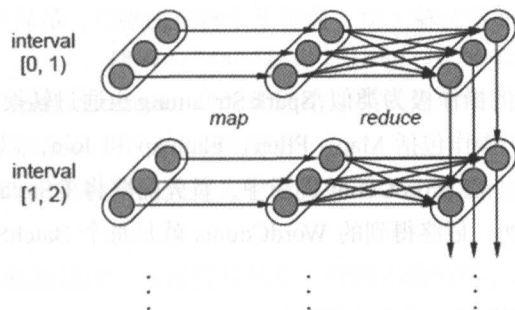


图 5.11

扩展性与吞吐量：Spark 目前在 EC2 上已经能够线性扩展到 100 个节点（每个节点 4Core），可以以数秒的延迟处理 6GB/s 的数据量（60Mrecords/s），其吞吐量也比流行的 Storm 高 2~5 倍。图 5.12 是 Berkeley 利用 WordCount 和 Grep 两个用例所做的测试，在 Grep 这个测试中，Spark Streaming 中每个节点的吞吐量是 670krecords/s，而 Storm 是 115krecords/s。

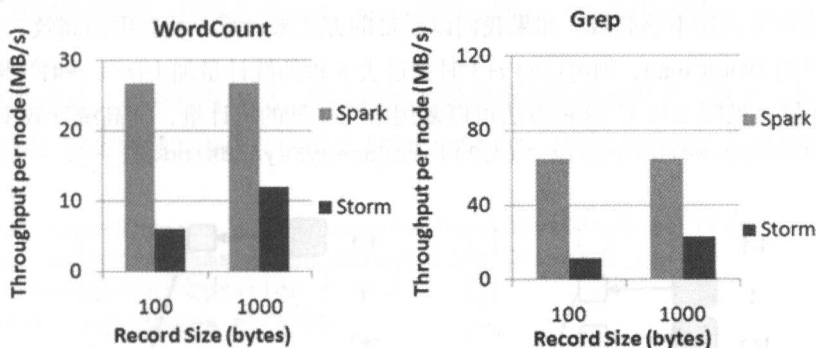


图 5.12

2. Spark Streaming 编程举例

Spark Streaming 的编程和 Spark 的编程如出一辙，对于编程的理解也非常类似。对于 Spark 来说，编程就是对 RDD 的操作；而对于 Spark Streaming 来说，编程就是对 DStream 的操作。下面通过一个大家熟悉的 WordCount 的例子来说明 Spark Streaming 中的输入操作、转换操作和输出操作。

初始化：在进行 DStream 操作之前，需要对 Spark Streaming 进行初始化，生成 StreamingContext。比较重要的是第一个和第三个参数，第一个参数用来指定 Spark Streaming 运行的集群地址，而第三个参数用来指定 Spark Streaming 运行时的 Batch 窗口大小。在这个例子中，就是将 1 秒钟的输入数据进行一次 SparkJob 处理。

```
val ssc=new StreamingContext("Spark://...", "WordCount", Seconds(1), [Homes], [Jars])
```

输入操作：目前 Spark Streaming 支持丰富的输入接口，大致分为两类：一类是磁盘输入，如以 BatchSize 作为时间间隔监控 HDFS 文件系统的某个目录，将目录中内容的变化作为 Spark Streaming 的输入；另一类就是网络流的方式，目前支持 Kafka、Flume、Twitter 和 TCP Socket。在 WordCount

例子中，假定通过网络 Socket 作为输入流，监听某个特定的端口，最后得出输入 DStream (lines)。

```
val lines=ssc.socketTextStream("localhost",8888)
```

转换操作：与 SparkRDD 的操作极为类似，Spark Streaming 也通过转换操作将一个或多个 DStream 转换成新的 DStream。常用的操作包括 Map、Filter、Flatmap 和 Join，以及需要进行 Shuffle 操作的 groupByKey/reduceByKey 等。在 WordCount 例子中，首先需要将 DStream (lines) 切分成单词，然后将相同单词的数量进行叠加，最终得到的 WordCounts 就是每个 BatchSize 的 (单词，数量) 中间结果。

```
val words=lines.flatMap(_.split(" "))
val wordCounts=words.map(x=>(x,1)).reduceByKey(_+_)
```

另外，Spark Streaming 有特定的窗口操作，窗口操作涉及两个参数：一个是滑动窗口的宽度 (WindowDuration)；另一个是窗口滑动的频率 (SlideDuration)，这两个参数必须是 BatchSize 的倍数。例如，以过去 5 秒为一个输入窗口，每 1 秒统计一下 WordCount，那么我们会将过去 5 秒的每 1 秒的 WordCount 都进行统计，然后进行叠加，得出这个窗口中的单词统计。

```
val wordCounts=words.map(x=>(x,1)).reduceByKeyAndWindow(_+_ ,Seconds(5s),seconds(1))
```

但是上面这种方式还不够高效。如果我们以增量的方式来计算，就会更加高效。例如，计算 $t+4$ 秒过去 5 秒窗口的 WordCount，则可以将 $t+3$ 时刻过去 5 秒的统计量加上 $[t+3, t+4]$ 的统计量，再减去 $[t-2, t-1]$ 的统计量 (见图 5.13)。这种方法可以复用中间 3 秒的统计量，提高统计效率。

```
val wordCounts=words.map(x=>(x,1)).reduceByKeyAndWindow(_+_ ,_-_,Seconds(5s),seconds(1))
```

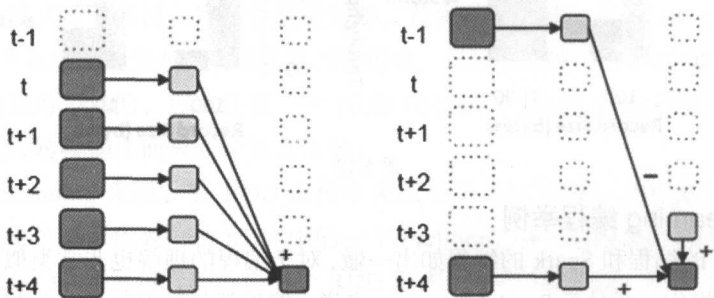


图 5.13

输出操作：对于输出操作，Spark 提供了将数据打印到屏幕及输出到文件中的功能。在 WordCount 中，我们将 DStreamwordCounts 输出到 HDFS 文件中。

```
wordCounts=saveAsHadoopFiles("WordCount")
```

启动：经过上述操作，Spark Streaming 还没有开始工作，还需要调用 Start 操作，Spark Streaming 才开始监听相应的端口，然后获取数据，并进行统计。

```
ssc.start()
```

3. Spark Streaming 深入

下面深入介绍 Spark Streaming 的内部机制、DStream、初始化、数据处理过程等内容。

1) DStream 的概念^①

① DStream

如果要用一句话来概括 Spark Streaming 的处理思路，那就是“将连续的数据持久化、离散化，然后进行批量处理”。下面详细分析一下这么做的原因。

- 数据持久化：将从网络上接收到的数据先暂时存储下来，为事件处理出错时的事件重演提供可能。
- 离散化：数据源源不断地涌进，永远没有尽头。既然不能穷尽，那么就将其按时间分片。比如采用 1 分钟为时间间隔，那么，在连续的 1 分钟内收集到的数据集中存储在一起。
- 批量处理：将持久化下来的数据分批进行处理，处理机制套用之前的 RDD 模式。

DStream 可以说是对 RDD 的又一层封装。如果打开 DStream.scala 和 RDD.scala 文件，则可以发现，几乎 RDD 上的所有 Operation 在 DStream 中都有相应的定义。作用于 DStream 上的 Operation 分为两类：

- Transformation。
- Output，表示输出结果，目前支持 print、saveAsObjectFiles、saveAsTextFiles、saveAsHadoopFiles。

② DStreamGraph

有输入就要有输出，如果没有输出，那么前面所做的所有操作就失去了意义。如何将这些输入和输出进行绑定呢？这个问题的解决就依赖于 DStreamGraph。DStreamGraph 记录输入的 Stream 和输出的 Stream。

```
private val inputStreams = new ArrayBuffer[InputDStream[_]]()
private val outputStreams = new ArrayBuffer[DStream[_]]()
var rememberDuration: Duration = null
var checkpointInProgress = false
```

outputStreams 中的元素是在有 Output 类型的 Operation 作用于 DStream 上时自动添加到 DStreamGraph 中的。

outputStream 区别于 inputStream 的一个重要地方就是会重载 generateJob。

2) 初始化流程

StreamingContext 是 Spark Streaming 初始化的入口点，其主要功能是根据入参来生成 JobScheduler，如图 5.14 所示。

① 设定 InputStream

如果数据源来自 Socket，则使用 socketStream；如果数据源来自不断变化的文件，则使用 fileStream。

^① 参考 <http://www.cnblogs.com/hseagle/p/3673142.html>。

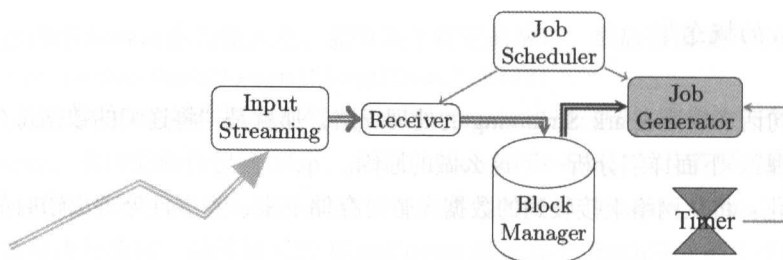


图 5.14

② 提交运行

```
StreamingContext.start()
```

3) 数据处理

数据处理流程如图 5.15 所示。

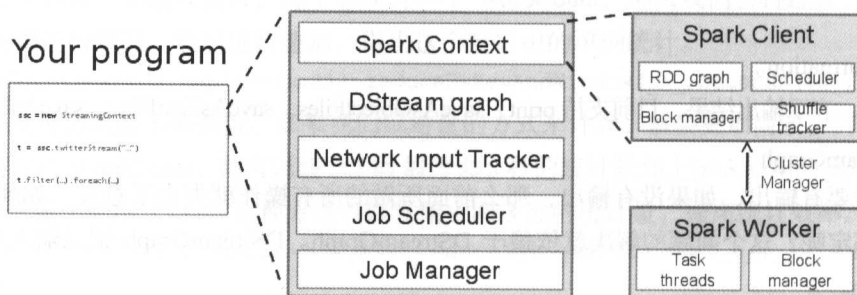


图 5.15

以 `socketStream` 为例,数据源来自 `Socket`。`SocketInputDStream` 启动一个线程,该线程使用 `receive` 函数来接收数据。

```

def receive() {
  var socket:Socket = null
  try {
    logInfo("Connecting to " + host + ":" + port)
    socket = new Socket(host,port)
    logInfo("Connected to " + host + ":" + port)
    val iterator = bytesToObjects(socket.getInputStream())
    while(!isStopped && iterator.hasNext) {
      store(iterator.next())
    }
    logInfo("Stopped receiving")
    restart("Retrying connecting to " + host + ":" + port)
  } catch {
    case e:java.net.ConnectException =>
      restart("Error connecting to " + host + ":" + port,e)
    case t:Throwable =>
      restart("Error receiving data",t)
  } finally {

```

```

if (socket != null) {
  socket.close()
  logInfo("Closed socket to " + host + ":" + port)
}
}
}
}

```

接收到的数据会先被存储起来，存储最终会调用到 `BlockManager.scala` 中的函数。那么，`BlockManager` 是如何被传递到 `StreamingContext` 的呢？是利用 `SparkEnv` 传入的，注意 `StreamingContext` 构造函数的入参，如图 5.16 所示。

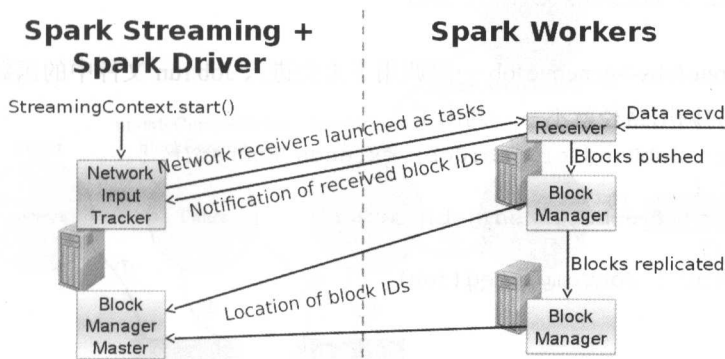


图 5.16

处理定时器

数据的存储是被 `Socket` 触发的。那么，已经存储的数据被真正处理又是被什么触发的呢？

在初始化 `StreamingContext` 的时候，我们指定了一个时间参数，可以用这个参数构造相应的重复定时器，一旦定时器超时，就调用 `generateJobs` 函数。

```

private val timer = new RecurringTimer(clock, ssc.graph.batchDuration.milliseconds,
longTime => eventActor ! generateJobs(new Time(longTime)), "JobGenerator")

```

(1) 事件处理函数。

```

/** Processes all events */
private def processEvent(event: JobGeneratorEvent) {
  logDebug("Got event " + event)
  event match {
    case GenerateJobs(time) => generateJobs(time)
    case ClearMetadata(time) => clearMetadata(time)
    case DoCheckpoint(time) => doCheckpoint(time)
    case ClearCheckpointData(time) => clearCheckpointData(time)
  }
}

```

(2) `generateJobs` 函数。

```

private def generateJobs(time: Time) {
  SparkEnv.set(ssc.env)

```

```

Try(graph.generateJobs(time)) match {
  case Success(jobs) =>
    val receivedBlockInfo = graph.getReceiverInputStreams.map { stream =>
      val streamId = stream.id
      val receivedBlockInfo = stream.getReceivedBlockInfo(time)
      (streamId, receivedBlockInfo)
    }.toMap
    jobScheduler.submitJobSet(JobSet(time, jobs, receivedBlockInfo))
  case Failure(e) =>
    jobScheduler.reportError("Error generating jobs for time " + time, e)
}
eventActor ! DoCheckpoint(time)
}

```

(3) 函数 `generateJobs->generateJob` 一路调用下去会进入 `Job.run` 文件中的函数，在 `job.run` 中调用 `sc.runJob`，具体的调用路径就不一一列出了。

```

private class JobHandler(job:Job) extends Runnable {
  def run() {
    eventActor ! JobStarted(job)
    job.run()
    eventActor ! JobCompleted(job)
  }
}

```

(4) 在 `DStream.generateJob` 函数中定义了 `jobFunc`，也就是在 `job.run` 中用到的 `jobFunc`。

```

private[streaming] def generateJob(time:Time):Option[Job] = {
  getOrCompute(time) match {
    case Some(rdd) => {
      val jobFunc = () => {
        val emptyFunc = { (iterator:Iterator[T]) => {} }
        context.sparkContext.runJob(rdd, emptyFunc)
      }
      Some(new Job(time, jobFunc))
    }
    case None => None
  }
}

```

在这个流程中，`DStreamGraph` 起到了非常关键的作用，类似于 `TridentStorm` 中的 `Graph`。

在 `generateJob` 过程中，`DStream` 会通过调用 `compute` 函数生成相应的 `RDD`，`SparkContext` 则是将基于 `RDD` 的抽象转换成多个 `Stage` 来执行。

`StreamingContext` 中的一个重要转换就是 `DStream` 到 `RDD` 的转换，而 `SparkContext` 中的一个重要转换则是 `RDD` 到 `Stage` 及 `Task` 的转换。在这两个不同的抽象类中，要注意 `getOrCompute` 和 `compute` 函数的实现。

4) DStream 处理的容错性分析^①

在流数据的处理过程中，为了保证处理结果的可信度（不能多算，也不能漏算），需要做到对所有的输入数据有且仅有一次处理。在 Spark Streaming 的处理机制中，不能多算比较容易理解。那么，它又是如何做到即使数据处理节点被重启，在重启之后这些数据也会被再次处理的呢？

① 数据接收过程

下面来看一下代码实现层面，一个是控制层面（Control Panel），另一个是数据层面（Data Panel）。Spark Streaming 的数据接收过程的控制层面大致如图 5.17 所示。

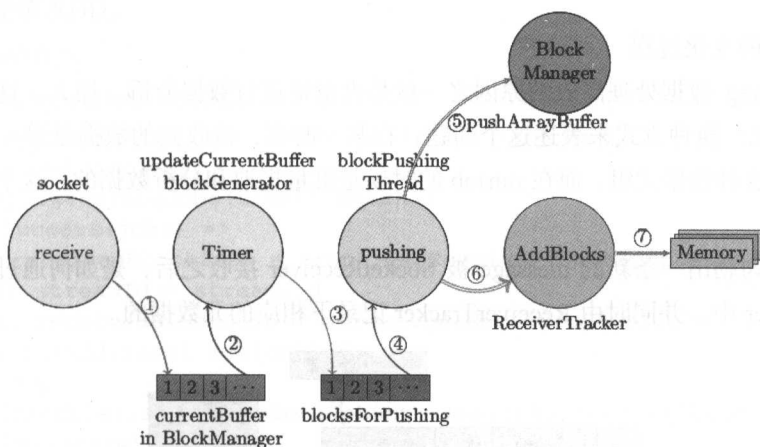


图 5.17

简要分析一下图 5.17 的含义。

- 数据真正被接收到发生在 `SocketReceiver.receive` 函数中，将接收到的数据放入 `BlockGenerator.currentBuffer` 中。
- 在 `BlockGenerator` 中有一个重复定时器，处理函数为 `updateCurrentBuffer`。`updateCurrentBuffer` 函数将当前 `Buffer` 中的数据封装为一个新的 `Block`，放入 `blocksForPushing` 队列中。
- 同样，在 `BlockGenerator` 中有一个 `BlockPushingThread`，其职责就是不停地将 `blocksForPushing` 队列中的成员通过 `pushArrayBuffer` 函数传递给 `BlockManager`，让 `BlockManager` 将数据存储到 `MemoryStore` 中。
- `pushArrayBuffer` 还会将已经由 `BlockManager` 存储的 `Block` 的 `ID` 传递给 `ReceiverTracker`，`ReceiverTracker` 会将存储的 `blockId` 放入对应 `StreamId` 的队列中。

`socket.receive`→`receiver.store`→`pushSingle`→`blockgenerator.updateCurrentBuffer`→`blockgenerator.keepPushBlocks`→`pushArrayBuffer`→`ReceiverTracker.addBlocks`

`pushArrayBuffer` 函数的定义如下：

```
def pushArrayBuffer(
  arrayBuffer:ArrayBuffer[_],
```

^① 参考 <http://www.cnblogs.com/hseagle/p/3673139.html>。

```

        optionalMetadata:Option[Any],
        optionalBlockId:Option[StreamBlockId]
    ) {
        val blockId = optionalBlockId.getOrElse(nextBlockId)
        val time = System.currentTimeMillis
        blockManager.put(blockId,arrayBuffer.asInstanceOf[ArrayBuffer[Any]],
            storageLevel,tellMaster = true)
        logDebug("Pushed block " + blockId + " in " + (System.currentTimeMillis - time) +
            " ms")
        reportPushedBlock(blockId,arrayBuffer.size,optionalMetadata)
    }

```

② 数据结构的变化过程

Spark Streaming 数据处理高效的原因之一就是批量地进行数据分析，那么，这些批量的数据是如何聚集起来的呢？换种方式来表述这个问题：在某一时刻，接收到的数据是单一的，也就是最多只能组成<t,data>这种数据元组，而在 runJob 的时候是批量提取和分析数据的，这个批量数据是在什么时候组成的呢？

图 5.18 大致勾勒出一条新的 message 被 SocketReceiver 接收之后，是如何通过一系列的处理而放入 BlockManager 中，并同时由 ReceiverTracker 记录下相应的元数据的。

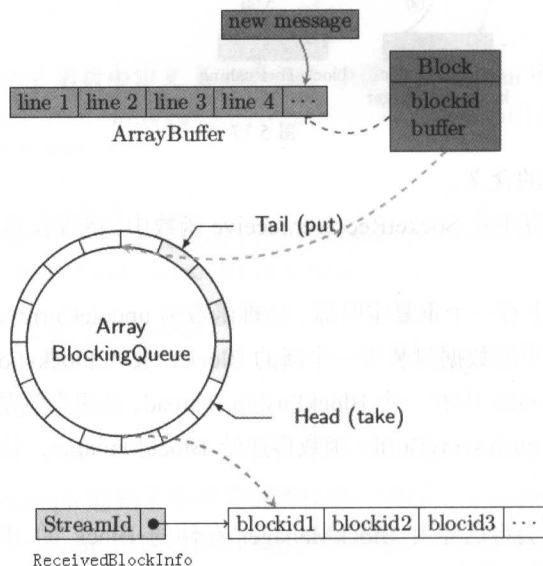


图 5.18

- 首先，new message 被放入 blockManager.currentBuffer 中。
- 定时器超时处理过程，将整个 currentBuffer 中的数据打包成一条 Block，放入 ArrayBlockingQueue 中，该数据结构支持 FIFO。
- keepPushingBlocks 将每条 Block（Block 中包含时间戳、接收到的原始数据）由 BlockManager 保存，同时通知 ReceiverTracker 已经将哪些 Block 存储到 BlockManager 中。

- ReceiverTracker 将每个 Stream 接收到但还没有进行处理的 Block 放入 ReceivedBlockInfo 中，其为一个 Hash map。在后面的 generateJobs 中，会从 ReceivedBlockInfo 中提取数据生成相应的 RDD。

③ 数据处理过程

数据处理中最重要的函数就是 generateJobs，该函数会引发下述的函数调用过程，具体的代码就不一一罗列了。

- jobgenerator.generateJobs->dstreamgraph.generateJobs->dstream.generateJob->getOrCompute->compute 生成 RDD。
- job 调用 job.func。

JobGenerator.generateJobs 函数定义如下：

```
private def generateJobs(time:Time) {
  SparkEnv.set(ssc.env)
  Try(graph.generateJobs(time)) match {
    case Success(jobs) =>
      val receivedBlockInfo = graph.getReceiverInputStreams.map { stream =>
        val streamId = stream.id
        val receivedBlockInfo = stream.getReceivedBlockInfo(time)
        (streamId,receivedBlockInfo)
      }.toMap
      jobScheduler.submitJobSet(JobSet(time,jobs,receivedBlockInfo))
    case Failure(e) =>
      jobScheduler.reportError("Error generating jobs for time " + time,e)
  }
  eventActor ! DoCheckpoint(time)
}
```

先来看看数据处理阶段是如何与上述接收阶段中存储下来的数据相联系的。

假设上一次进行 RDD 处理发生在时间点 t_1 ，现在是时间点 t_2 ，那么在 $\langle t_2, t_1 \rangle$ 之间有哪些 Blocks 没有被处理呢？想必你已经知道答案了，没有被处理的 Blocks 全部保存在 ReceiverTracker 的 receiverBlockInfo 之中。

在 generateJob 时，每个 DStream 都会调用 getReceivedBlockInfo。getReceivedBlockInfo 和 ReceiverTracker 中的 receivedBlockInfo 的关系，看数据输入的源头 ReceiverInputDStream 中的 getReceivedBlockInfo 是如何定义的，代码列举如下：

```
private[streaming] def getReceivedBlockInfo(time:Time) = {
  receivedBlockInfo(time)
}
```

那么，此处的 receivedBlockInfo(time)是从何而来的呢？这要看 ReceivedInputDStream 中的 compute 函数实现，如下：

```
override def compute(validTime:Time):Option[RDD[T]] = {
  // If this is called for any time before the start time of the context,
  // then this returns an empty RDD. This may happen when recovering from a
```

```
// master failure
if (validTime >= graph.startTime) {
  val blockInfo = ssc.scheduler.receiverTracker.getReceivedBlockInfo(id)
  receivedBlockInfo(validTime) = blockInfo
  val blockIds = blockInfo.map(_.blockId.asInstanceOf[BlockId])
  Some(new BlockRDD[T](ssc.sc, blockIds))
} else {
  Some(new BlockRDD[T](ssc.sc, Array[BlockId]()))
}
}
```

至此，终于看到 ReceiverTracker 中的 getReceivedBlockInfo 被调用了，也就是说将接收阶段的数据和目前处理阶段的输入通道打通了。

函数调用路径为从 generateJobs 到 sparkcontext.submitJobs。这个时候要注意，注册在 DStreamGraph 中的 OutputStream 上的操作会引发 SparkContext.runJobs 被调用。我们以 print 函数为例来看一下调用过程。

```
def print() {
  def foreachFunc = (rdd: RDD[T], time: Time) => {
    val first11 = rdd.take(11)
    println ("-----")
    println ("Time:" + time)
    println ("-----")
    first11.take(10).foreach(println)
    if (first11.size > 10) println("...")
    println()
  }
  new ForEachDStream(this, context.sparkContext.clean(foreachFunc)).register()
}
```

注意，rdd.take 会引发 runJob 调用，我们可以看看其定义中调用 runJob 的片段。

```
val left = num - buf.size
val p = partsScanned until math.min(partsScanned + numPartsToTry, totalParts)
val res = sc.runJob(this, (it: Iterator[T]) => it.take(left).toArray, p,
allowLocal = true)
res.foreach(buf += _.take(num - buf.size))
partsScanned += numPartsToTry
}
```

小结一下数据处理过程：

- 以 time 为关键字取出在此时间之前加入的所有 blockIds。
- 在真正提交运行的时候，rdd 中的 BlockFetcher 以 blockId 为关键字到 BlockManagerMaster 中获取真正的数据，即从 Socket 上接收到的原始数据。

④ 容错处理

在 JobGenerator.generateJobs 函数运行的最后会发出 DoCheckpoint 通知，该通知会让相应的 Actor 将 DStreamCheckpointData 写入 HDFS 文件中。我们来看一看为什么需要写入 CheckpointData，以及哪些内容是包含在 CheckpointData 中的。

在数据处理一节，我们已经分析出，在 `generateJobs` 的时候会生成多个 `Jobs`，它们会通过 `sparkcontext.runJob` 接口发送到 `Cluster` 中被真正执行。

假设在如图 5.19 所示的 t_2 时刻，Worker 挂掉了，挂掉的 Worker 直到 t_3 时刻才完全恢复。由于挂掉的原因，上一次 `generateJobs` 生成的 `Job` 不一定被完全处理了（也许有些已经处理了，有些还没有处理），所以需要重新提交一次。这样就可能导致针对同一批数据有重复处理的情况发生，从而无法达到 `exactly-once` 的语义效果。

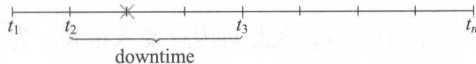


图 5.19

另外，在 $\langle t_2, t_3 \rangle$ 这段时间内，没有新的数据被接收，所以 Spark Streaming 的 `SocketReceiver` 适合用来充当 Client 侧而不是 Server 侧。`SocketReceiver` 读取到的数据应该存储在一个具有冗余备份机制的内存数据库或缓存队列里，如 `Kafka`。对于这个问题，Spark Streaming 是无法解决的。

⑤ CheckpointData

`Checkpoint` 的成员变量有哪些呢？我们看一看其结构定义就清楚了。

```
val master = ssc.sc.master
val framework = ssc.sc.appName
val sparkHome = ssc.sc.getSparkHome.getOrElse(null)
val jars = ssc.sc.jars
val graph = ssc.graph
val checkpointDir = ssc.checkpointDir
val checkpointDuration = ssc.checkpointDuration
val pendingTimes = ssc.scheduler.getPendingTimes().toArray
val delaySeconds = MetadataCleaner.getDelaySeconds(ssc.conf)
val sparkConfPairs = ssc.conf.getAll
```

`generatedRDDs` 是被包含在 `Graph` 里面的，所以不要突然之间惊慌失措，发觉没有将 `generatedRDDs` 保存起来。

`Checkpoint` 的数据是通过 `CheckpointWriteHandler` 真正写入 HDFS 的，通过 `CheckpointReader` 读入。`CheckpointReader` 在重启的时候会被用到，判断是第一次干净地启动还是因错误而重启，判断的依据都在 `cp` 这个变量里。

为了在重启之后自动检查并载入相应的 `Checkpoint` 数据，在创建 `StreamingContext` 的时候不能简单地通过调用 `new StreamingContext` 来完成，而是要调用 `getOrCreate` 函数。示例代码如下：

```
// Function to create and setup a new StreamingContext
def functionToCreateContext():StreamingContext = {
    val ssc = new StreamingContext(...) // new context
    val lines = ssc.socketTextStream(...) // create DStreams
    ...
    ssc.checkpoint(checkpointDirectory) // set checkpoint directory
    ssc
}
// Get StreaminContext from checkpoint data or create a new one
val context = StreamingContext.getOrCreate(checkpointDirectory, functionTo
```

```
CreateContext(_)
// Do additional setup on context that needs to be done,
// irrespective of whether it is being started or restarted
context. ...
// Start the context
context.start()
context.awaitTermination()
```

4. 优化内存使用

Spark 将所有的处理都放到内存中进行，因此对内存要求很高。为了高效地运行，必须优化内存的使用。通常有以下几种优化思路：

(1) 控制 BatchSize。Spark Streaming 会把 Batch 窗口内接收到的所有数据存放在 Spark 内部的可用内存区域中，因此必须确保当前节点 Spark 的可用内存至少能够容纳这个 Batch 窗口内的所有数据，否则必须增加新的资源以提高集群的处理能力。

(2) 及时清理不再使用的数据。Spark Streaming 会将接收到的数据全部存储于内部的可用内存区域中，因此，对于处理过的不再需要的数据应及时清理，以确保 Spark Streaming 有足够的可用内存空间。可以通过设置合理的 spark.cleaner.ttl 时长来及时清理超时的无用数据。

(3) 观察及适当调整 GC 策略。GC 会影响 Job 的正常运行，延长 Job 的执行时间，引起一系列不可预料的问题。观察 GC 的运行情况，采取不同的 GC 策略，以进一步减小内存回收对 Job 运行的影响。

5.4.3 融合框架^①

批处理和流处理，从业务上看对时间的要求不同，采用了不同的处理技术，所以通常是两套完全不同的系统，但是往往又有一部分实时和批处理的数据是重叠的。另外，从全系统角度看，过多的系统带来更大的维护难度，所以业界一直在探索批处理系统和流处理系统的融合与统一。

在当前的过渡阶段，流行的 Lambda 架构就是流处理和批处理相结合的系统，如图 5.20 所示。

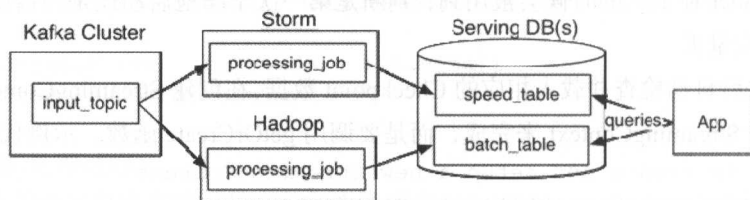


图 5.20

这种方式对于不可变的记录序列处理得很好，将这些不可变的记录截获后并行地送入批处理系统和流处理系统，实现两次逻辑转换（一次是在批处理系统，另一次是在流处理系统），然后在查询时将两个系统的处理结果混合在一起，产生一个完整的响应结果。

^① 质疑 Lambda 架构，<http://www.jdon.com/46506>。

Lambda 架构带来的主要问题是维护两套系统的困难，所以统一的流处理系统是否有可能？未来的流处理能力是否会提高到支持所有场景？Twitter 使用 Samza 实现重复 reprocessing 来处理批数据，实现流和批处理的统一，它们将这种统一的架构叫作 Kappa Architecture，如图 5.21 所示。

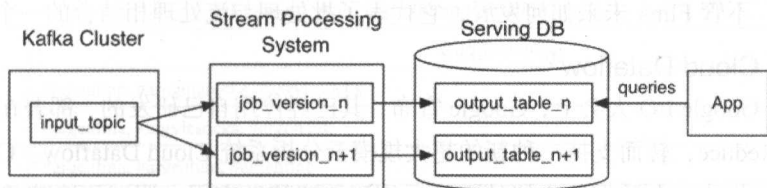


图 5.21

1. Flink^①

Flink 起源于德国，是 Apache 的顶级项目 (flink.apache.org)，如图 5.22 所示。相比 Spark，其最主要的特点是原生流系统，而不是 Spark Streaming 通过 Mini-Batch 模拟流处理。其他的一些独特优势如下。

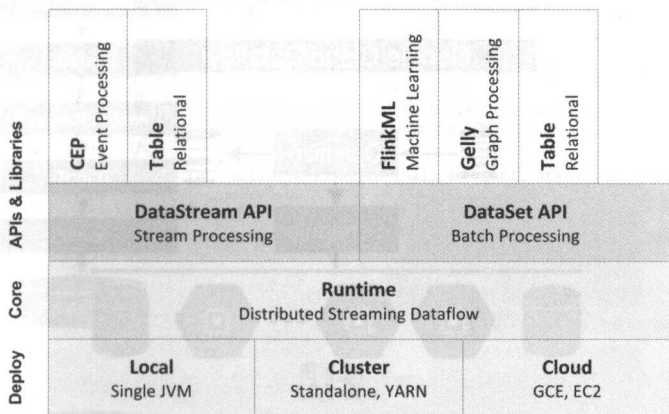


图 5.22

(1) 自动对处理流程的优化：如逻辑优化、物理优化等。真正的执行计划并非和用户所写的 code 一致（在结果相同的情况下，根据处理数据的特性和集群的特性通过优化得到最优的流程）。而 Spark 的执行计划与用户所写的 code 一致。

(2) 自动对迭代计算的优化：对全迭代（Spark 仅有全迭代）的优化 + 内置的增量式迭代（Delta Iteration）。对于迭代计算，Flink 会包装成闭环的流程。这与传统的将迭代计算看作数个独立任务有很大的不同。

(3) Out-of-Core 计算机制：灵活使用本地的硬盘和内存资源，无须用户自己调整。

(4) Pipeline 机制：计算数据及时，直接发送到下一个任务；下一个任务不必等待上一个任务执行完毕才开始执行。

① Flink 官网 <http://flink.apache.org/>。

(5) 同时支持 Batch 处理和 Streaming 处理。得益于 Pipeline 机制, Flink 实际上是基于 Streaming 机制的 Batch 处理引擎。

Spark 和 Flink 之间是直接竞争关系, Spark 的 Tungsten 项目很多是直接对标 Flink 的, 如内存管理等。总的来说, 不管 Flink 未来如何发展, 它代表了批处理与流处理相结合的一个很好的尝试。

2. Google Cloud Dataflow^①

在 2014 年的 Google I/O 大会上, Google 宣布, 其已经停用自己研发的、部署在服务器上、用以分析数据的 MapReduce, 转而支持一种新的超大规模云分析系统 Cloud Dataflow。Cloud Dataflow 是一种构建、管理和优化复杂数据流水线的方法, 用于构建移动应用, 调试、追踪和监控产品级云应用。它采用了 Google 内部的 Flume 和 MillWheel 技术, 其中 Flume 用于数据的高效并行化处理, 而 MillWheel 则用于互联网级别的带有很好的容错机制的流处理。其架构如图 5.23 所示。

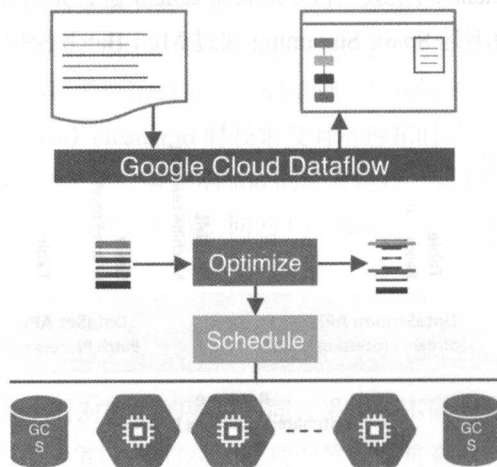


图 5.23

Dataflow 当前的 API 只有 Java 版本（其实 Flume 本身提供了 Java、C++、Python 等多种接口, MillWheel 也提供了 Java、C++ 的 API）。相比原生的 Map/Reduce 模型, Dataflow 有以下几个优点:

(1) 可以构建复杂的 Pipeline。在这里不妨引用 Google 云平台的产品营销总监 Brian Goldfarb 的话, “Cloud Dataflow 可以用于处理批量数据和流数据两种”。在一个世界性事件（如演讲中的世界杯事件）中, 需要实时分析上百万条 Twitter 数据。在流水线的一个阶段负责读取 tweet, 下一个阶段负责抽取标签, 另一个阶段对 tweet 进行分类（基于情感、正负面或者其他方面）, 下一个阶段过滤关键词, 等等。相比之下, Map/Reduce 这个用来处理大数据的较早模型, 处理这种实时数据已经力不从心, 而且也很难应用到这种很长、很复杂的数据流水线上。

(2) 无须手工配置和管理 MapReduce 集群。能够自动进行代码优化和资源调度, 使得开发者的主要精力可以放在业务逻辑本身。

① 十分钟了解分布式计算: Google Cloud Dataflow, <http://www.cnblogs.com/wei-li/p/Dataflow.html>。

(3) 支持从 Batch 到 Streaming 模式的无缝切换。

假设我们要根据用户在 Twitter 上产生的内容来实现一个 hashtags 自动补全的功能，如表 5.3 所示。

表 5.3

Example: Auto completing hashtags	
Prefix	Suggestions
ar	#argentina, #arugularocks, #argylesocks
arg	#argentina, #argylesocks, #argonauts
arge	#argentina, #argentum, #argentine

代码几乎和数据流一一对应，和单机程序的编写方式差别不大，如图 5.24 和图 5.25 所示。

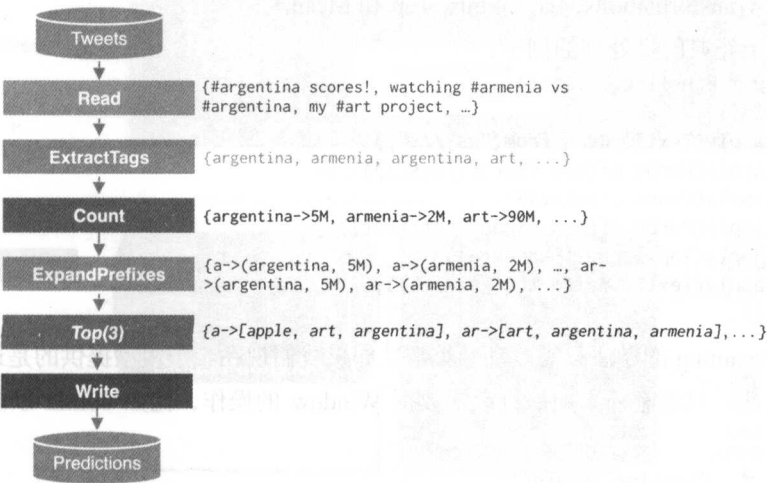


图 5.24

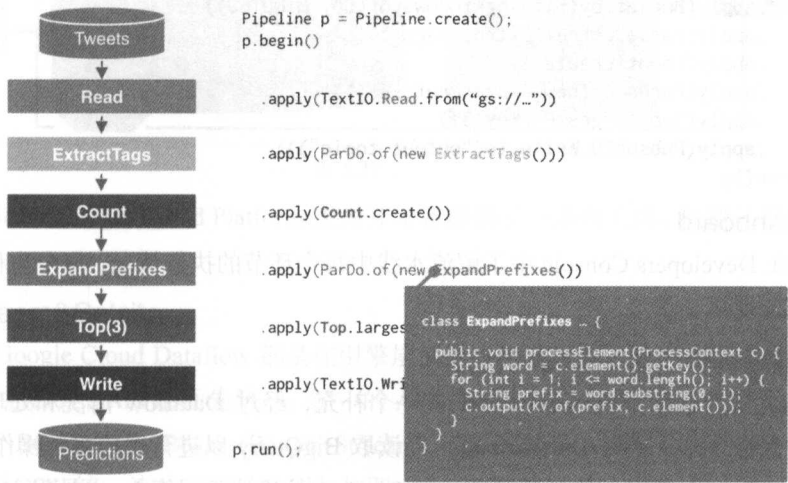


图 5.25

Dataflow 将数据抽象为一个 PCollection (Parallel Collection)。PCollection 可以是一个内存中的集合，从 Cloud Storage 读进来，从 BigQuerytable 中查询得到，从 Pub/Sub 中以流的方式读入，或者从用户代码中计算得到。

为了对 PCollection 进行处理，Dataflow 提供了许多 PTransforms (Parallel Transforms)，如 ParDo (Parallel Do) 对 PCollection 中的每个元素分别进行指定操作(类似于 MapReduce 中的 Map 和 Reduce 函数，或者 SQL 中的 WHERE)，GroupByKey 对一个 key-value pairs 的 PCollection 进行处理，将相同 key 的 pairs 聚集到一起(类似于 MapReduce 中的 Shuffle 步骤，或者 SQL 中的 GROUP BY 和 JOIN)。

此外，用户还可以将这些基本操作组合起来定义新的 Transformations。Dataflow 本身也提供了一些常用的组合 Transformations，如 Count、Top 和 Mean。

如下是一个经典的批处理的例子：

```
Pipeline p = Pipeline.create();
p.begin()
  .apply(TextIO.Read.from("gs://..."))
  .apply(ParDo.of(new ExtractTags()))
  .apply(Count.create())
  .apply(ParDo.of(new ExpandPrefixes()))
  .apply(Top.largestPerKey(3))
  .apply(TextIO.Write.to("gs://..."));
p.run();
```

转换为 Streaming 的做法只需改动数据源。如果我们现在希望模型提供的是最新的热词，那么考虑数据的时效性，只需额外添加一行设置数据 Window 的操作，比如 60min 以前的数据就不要了。

示例代码如下：

```
Pipeline p = Pipeline.create();
p.begin()
  .apply(PubsubIO.Read.from("input_topic"))
  .apply(Bucket.by(SlidingWindows.of(60, MINUTES)))
  .apply(ParDo.of(new ExtractTags()))
  .apply(Count.create())
  .apply(ParDo.of(new ExpandPrefixes()))
  .apply(Top.largestPerKey(3))
  .apply(PubsubIO.Write.to("output_topic"));
p.run();
```

3. Dashboard

还可以在 Developers Console 中了解流水线中每个环节的执行情况，每个流程框基本对应一行代码，如图 5.26 所示。

4. 生态系统

BigQuery 作为存储系统是 Dataflow 的一个补充，经过 Dataflow 清洗和处理过的数据，可以在 BigQuery 中保存下来，同时 Dataflow 也可以读取 BigQuery 以进行表连接等操作，如图 5.27 所示。如果想在 Dataflow 上使用一些开源资源(如 Spark 中的机器学习库)，也是很方便的。

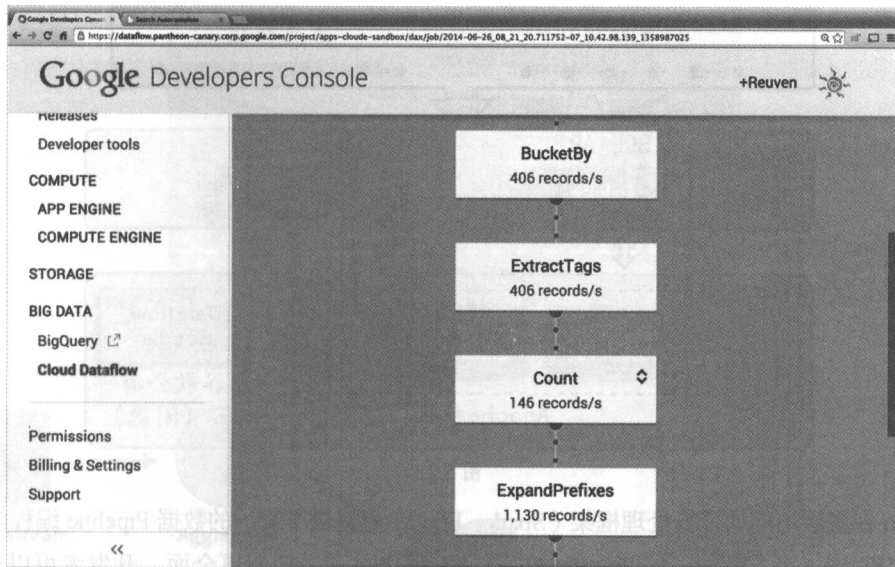


图 5.26

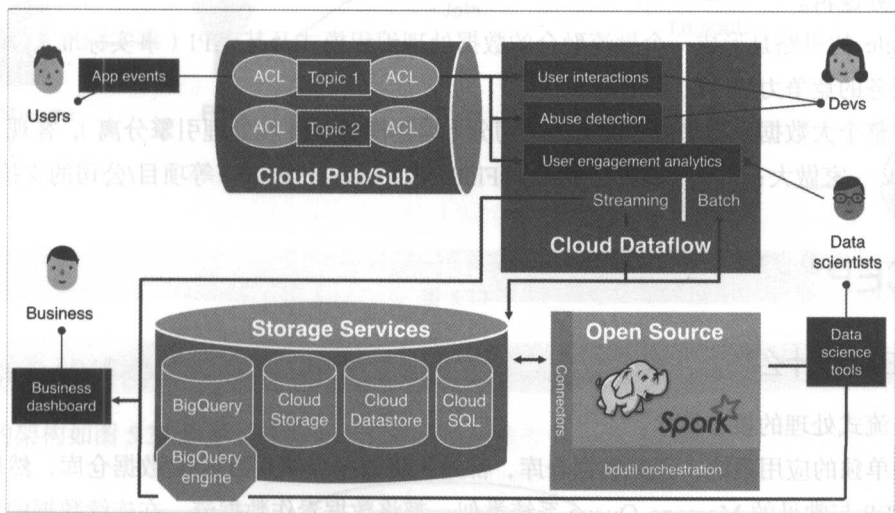


图 5.27

为了配合 Dataflow, Google Cloud Platform 还为开发者提供了一系列工具, 包括云保存、云调试、云追踪和云监控。

5. Google Beam&Calcite

Spark、Flink、Google Cloud Dataflow 都是在引擎层面统一批流处理, 这在短期内很难适应所有场景。所以另外一个思路是, API 层面先统一, 统一的 API 再对接底层多个引擎。Google 在 Dataflow 的基础上开源了一个 Beam 项目, Beam=(B)atch+Str(eam)。Beam 处于 API 层面, 而对于 SQL, 则可以采用 Calcite, 通过 SQL 统一流式和批量处理, 如图 5.28 所示。

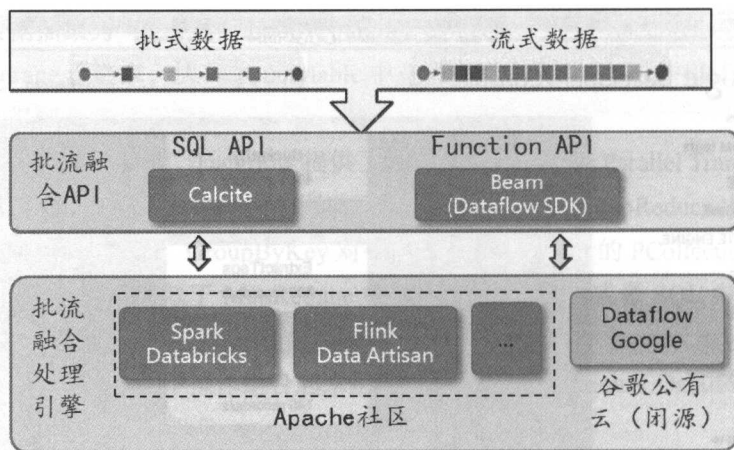


图 5.28

- Beam 在底层不同的计算处理框架（Spark、Flink）上提供了统一的数据 Pipeline 编程模型和 API。
- 其 SDK 和 API 定义相比，Spark、Flink 抽象层次更高、功能更全面，开发者可以聚焦于自身的业务逻辑。
- Google 的思路是形成一个批流融合的数据处理编程模式及其 API（事实标准），增加 Google 云服务的竞争力/品牌。
- 对于整个大数据生态来说，价值链被切分（编程模型 API 和处理引擎分离），客观上可以防止 Spark 一家做大；目前 Beam 已经得到 Flink、Cloudera、Paypal 等项目/公司的支持。

5.5 CEP^①

5.5.1 CEP 是什么^②

CEP 是流式处理的核心技术。

不管是单独的应用系统，还是数据仓库，都是先将数据存储到数据库/数据仓库，然后再处理或查询。而 CEP 与常见的 Message Queue 系统类似，都将数据看作数据流，在连续数据的快速移动过程中进行分析处理。这样的方式不需要很大的数据加载，完全可以在内存中进行，从而能够快速产生结果，如图 5.29 所示。

业务事件可能很复杂，在各种不同的数据流中源源不断地产生各种类型的事件。不仅需要对这些业务事件进行复杂的计算，如过滤、关联、聚合等，同时还需要考虑这些业务事件出现的时间序列，最终才能产生有意义的事件，或触发业务流程。注意，这些计算的规则可能还会经常变化。

这类问题通常通过基于规则的推理机（规则引擎）来实现，如图 5.30 所示。

① CEP: Complex Event Processing, 复杂事件处理。

② CEP: 鱼与熊掌可以兼得, <http://www.cnblogs.com/holbrook/archive/2012/11/06/2756759.html>。

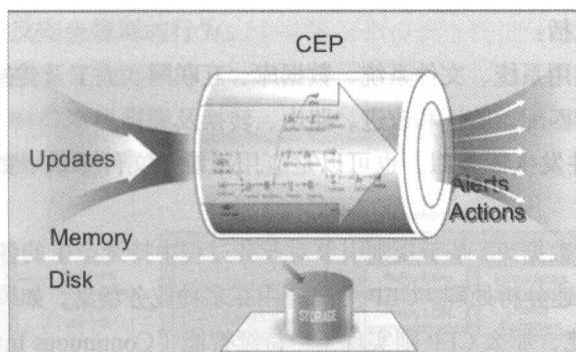


图 5.29

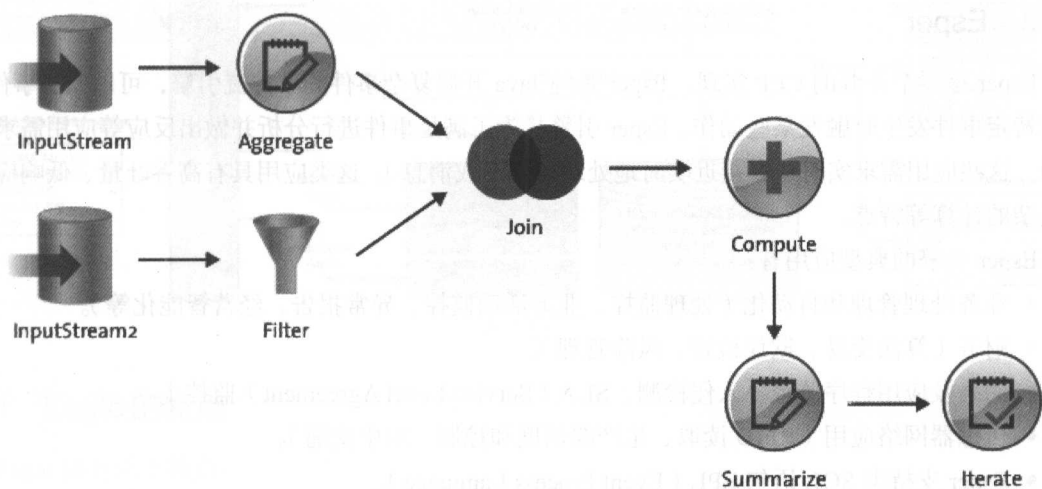


图 5.30

5.5.2 CEP 的架构

CEP 的架构如图 5.31 所示。

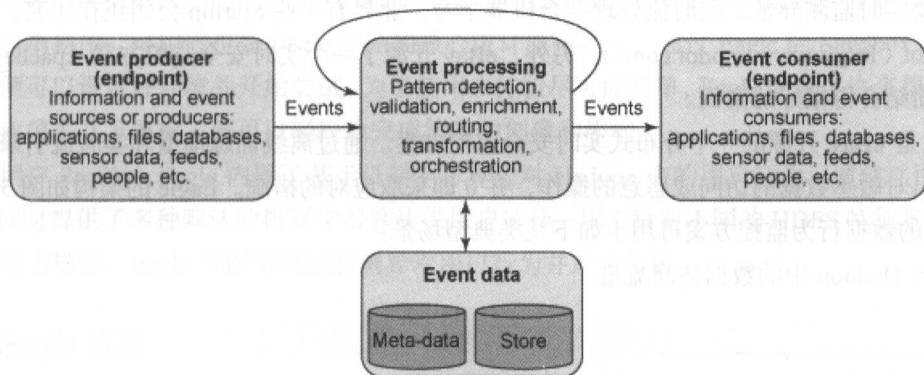


图 5.31

CEP 在逻辑上应该包括：

- 事件发生器通过应用系统、文件系统、数据库、互联网、人工及传感器产生事件。
- 事件处理器模式的匹配、验证和改进、路由、转换及编排。
- 事件消费者与事件发生器类似，也可以是应用系统、文件系统、数据库、互联网、人工界面等。

CEP 将数据看作一种数据流，基于规则引擎对业务过程中持续产生的各种事件进行复杂的处理，能够实现对连续数据的快速分析处理。CEP 可以应用在多种业务场景，如风险分析、程序化交易等。如果说 BI 实现了商业智能，那么 CEP 则实现了“持续智能（Continuous Intelligence）”。

5.5.3 Esper^①

Esper 是一个开源的 CEP 实现。Esper 是纯 Java 开源复杂事件和事件流引擎，可以监测事件流，并在特定事件发生时触发某些动作。Esper 引擎是为了满足事件进行分析并做出反应等应用需求而产生的。这些应用需求实时或者接近实时地处理事件（或消息）。这类应用具有高吞吐量、低响应时延和复杂的计算等特点。

Esper 引擎的典型应用有：

- 业务处理管理和自动化（处理监控、业务活动监控、异常报告、经营智能化等）。
- 财务（算法交易、欺诈检查、风险管理）。
- 网络及应用程序监控 [入侵检测、SLA（Service Level Agreement）监控]。
- 传感器网络应用（RFID 读取、生产线调度和控制、空中交通）。
- Esper 支持类 SQL 语句 EPL（Event Process Language）。

5.6 实时结合机器学习

机器学习主要应用于离线批量数据的处理，现实中也有一些场景需要结合机器学习，比如在安全领域需要实时监测异常。实时流处理结合机器学习，业界有一些 Startup 公司正在研究，比较有名的如 Anodot（<http://www.anodot.com/>）。另外，eBay 开源了一个实时安全监控方案 Apache Eagle^②，也是这方面结合得很好的案例。

Eagle 是 eBay 开源的一个分布式实时安全监控方案。通过离线训练模型和实时流引擎监控，能立即监测出对敏感数据的访问或恶意的操作，并立即采取应对的措施。Eagle 的架构如图 5.32 所示。

Eagle 的数据行为监控方案可用于如下几类典型场景：

- 监控 Hadoop 中的数据访问流量。

① ESP 介绍：<http://blog.csdn.net/luonanqin/article/details/9900295>。

② Apache Eagle：<http://eagle.apache.org/>。

- 检测非法入侵和违反安全规则的行为。
- 检测并防止敏感数据丢失和访问。
- 实现基于策略的实时检测和预警。
- 实现基于用户行为模式的异常数据行为检测。

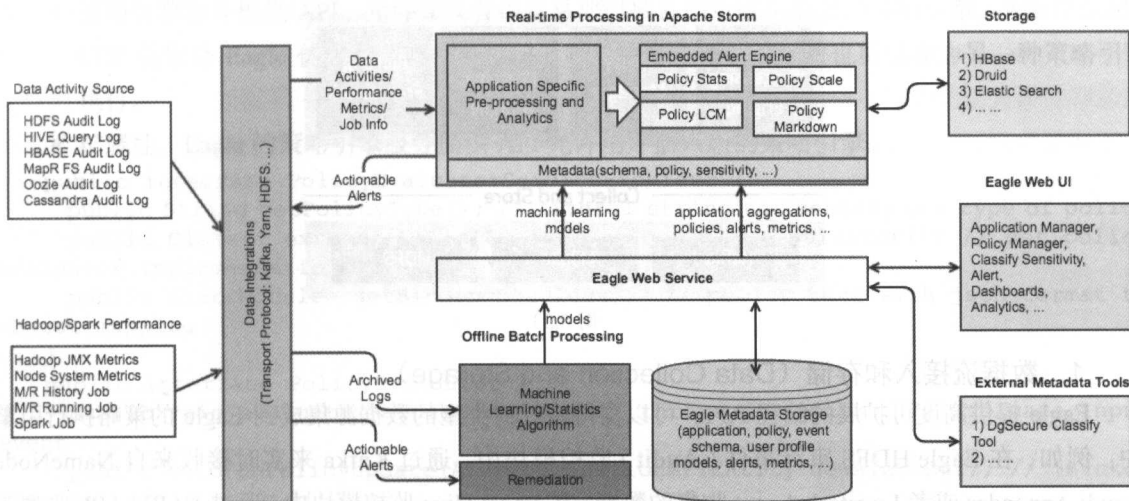


图 5.32

5.6.1 Eagle 的特点

Eagle 具有以下特点。

(1) 高实时: 由于充分理解了安全监控中高度实时和快速反应的重要性, 因此在设计 Eagle 之初, 尽可能地确保能在亚秒级别时间内产生告警, 一旦综合多种因素确定为危险操作, 立即采取措施阻止非法行为。

(2) 可伸缩: 在 eBay, Eagle 被部署在多个大型 Hadoop 集群上, 这些集群拥有数百 PB 的数据, 每天有 8 亿以上的数据访问时间, 因此 Eagle 必须具有处理海量实时数据的高度可伸缩能力。

(3) 简单易用: 可用性也是 Eagle 产品的核心设计原则之一。通过 Eagle 的 Sandbox, 使用者仅需数分钟便可以设置好环境并开始尝试。为了使用户体验尽可能简单, Eagle 内置了许多很好的例子, 只需简单地单击几下鼠标, 便可以轻松地完成策略的创建和添加。

(4) 用户 Profile: Eagle 内置了基于机器学习算法对 Hadoop 中的用户行为习惯建立用户 Profile 的功能。同时提供了多种默认的机器学习算法供用户选择, 用于针对不同的 HDFS 特征集进行建模。通过历史行为模型, Eagle 可以实时地监测异常用户行为并产生告警。

5.6.2 Eagle 概览

Eagle 概览如图 5.33 所示。

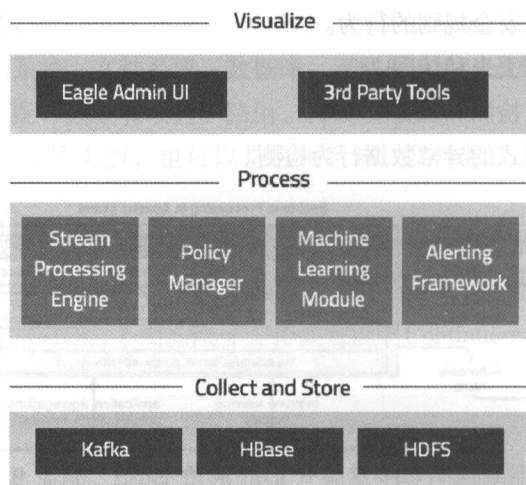


图 5.33

1. 数据流接入和存储 (Data Collection and Storage)

Eagle 提供高度可扩展的编程 API，可以支持将任何类型的数据源集成到 Eagle 的策略执行引擎中。例如，在 Eagle HDFS 审计事件 (Audit) 监控模块中，通过 Kafka 来实时接收来自 NameNode Log4j Appender 或者 Logstash Agent 收集的数据；在 Eagle Hive 监控模块中，通过 YARN API 收集正在运行 Job 的 Hive 查询日志，并保证较高的可伸缩性和容错性。

2. 数据实时处理 (Data Processing)

(1) 流处理 API (Stream Processing API)。Eagle 提供独立于物理平台而高度抽象的流处理 API，目前默认支持 Apache Storm，但是也允许扩展到其他任意流处理引擎，如 Flink 或者 Samza 等。该层抽象允许开发者在定义监控数据处理逻辑时，无须在物理执行层绑定任何特定的流处理平台，而只需通过复用、拼接和组装如数据转换、过滤、外部数据 Join 等组件，以实现满足需求的 DAG (有向无环图)。同时，开发者也可以很容易地以编程的方式将业务逻辑流程和 Eagle 策略引擎框架集成起来。Eagle 框架内部会将描述业务逻辑的 DAG 编译成底层流处理架构的原生应用，如 Apache Storm Topology 等，从而实现平台的独立。

以下是一个 Eagle 如何处理事件和告警的示例：

```
StormExecutionEnvironment env = ExecutionEnvironmentFactory.getStorm(config);
// storm env
StreamProducer producer = env.newSource(new KafkaSourcedSpoutProvider().
getSpout(config)).renameOutputFields(1) // declare kafka source
    .flatMap(new AuditLogTransformer()) // transform event
    .groupBy(Arrays.asList(0)) // group by 1st field
    .flatMap(new UserProfileAggregatorExecutor()); // aggregate one-hour data
by user
    .alertWithConsumer("userActivity","userProfileExecutor") // ML policy
evaluate
env.execute(); // execute stream processing and alert
```

(2) 告警框架 (Alerting Framework)。Eagle 告警框架由流元数据 API、策略引擎服务提供 API、策略 Partitioner API 及预警去重框架等组成。

- 流元数据 API。允许用户声明事件的 Schema，包括事件由哪些属性构成、每个属性的类型，以及当用户配置策略时如何在运行时动态解析属性的值等。
- 策略引擎服务提供 API。允许开发者很容易地以插件的形式扩展新的策略引擎。WSO2 Siddhi CEP 引擎是 Eagle 优先默认支持的策略引擎，同时机器学习算法也可以作为另一种策略引擎执行。
- 扩展性。Eagle 的策略引擎服务提供 API 允许用户插入新的策略引擎。

```
public interface PolicyEvaluatorServiceProvider {
    public String getPolicyType(); // literal string to identify one type of policy
    public Class<? extends PolicyEvaluator> getPolicyEvaluator(); // get policy
    evaluator implementation
    public List<Module> getBindingModules(); // policy text with json format to
    object mapping
}

public interface PolicyEvaluator {
    public void evaluate(ValuesArray input) throws Exception; // evaluate input
    event
    public void onPolicyUpdate(AlertDefinitionAPIEntity newAlertDef); // invoked
    when policy is updated
    public void onPolicyDelete(); // invoked when policy is deleted
}
```

- 策略 Partitioner API。允许策略在不同的物理节点上并行执行，同时允许用户自定义策略 Partitioner 类。这些功能使得策略和事件完全以分布式的方式执行。
- 可伸缩性 Eagle。通过支持策略的分区接口来实现大量的策略可伸缩并发地运行，如图 5.34 所示。示例代码如下：

```
public interface PolicyPartitioner extends Serializable {
    // method to distribute policies
    int partition(int numTotalPartitions, String policyType, String policyId);
}
```

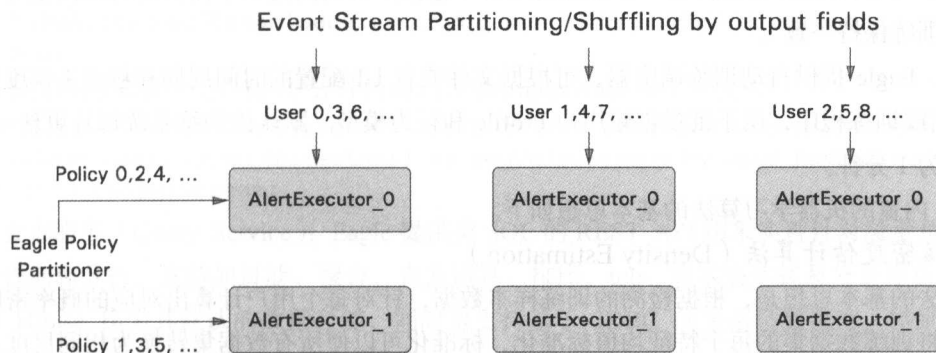


图 5.34

3. 机器学习模块

Eagle 支持根据用户在 Hadoop 平台上的历史使用行为习惯来定义行为模式或用户 Profile 的能力。拥有了这个功能，即便没有预先在系统中设置固定临界值，也可以智能地检测出异常行为。在 Eagle 中，用户 Profile 是通过机器学习算法生成的，用于在用户当前实时行为模式与其对应的历史模型模式存在一定程度的差异时识别用户行为是否异常。目前，Eagle 提供了两种算法来检测异常：特征值分解（Eigen-Value Decomposition）和密度估计（Density Estimation）。这两种算法从 HDFS 审计日志中读取数据，对数据进行分割、审查、交叉分析，周期性地为每个用户依次创建 Profile 行为模型。一旦模型生成，Eagle 的实时流策略引擎能够近乎实时地识别出异常，分辨当前用户的可疑行为。

图 5.35 简单描述了目前 Eagle 中用户 Profile 的离线训练建模和在线实时监测的数据流。

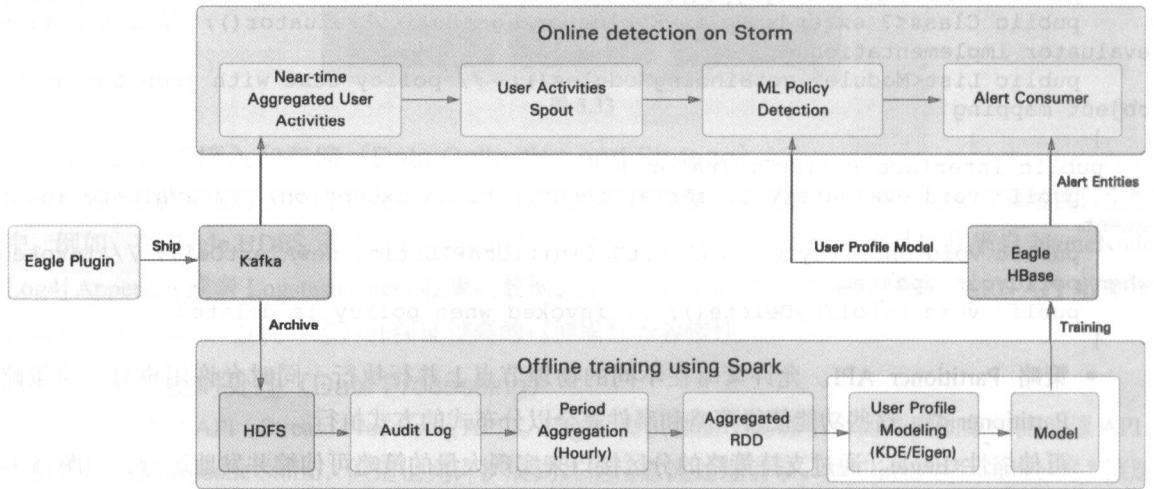


图 5.35

基于用户 Profile 的 Eagle 在线实时异常监测是根据 Eagle 的通用策略框架实现的，用户 Profile 只是被定义为 Eagle 系统中的一个策略而已。用户 Profile 的策略是通过继承自 Eagle 统一策略执行接口的机器学习 Evaluator 来执行的，其策略的定义中包括异常检测过程中需要的特征向量等（在线检测与离线训练保持一致）。

此外，Eagle 提供自动训练调度器，可根据文件或者 UI 配置的时间周期和粒度来调度这个基于 Spark 的离线训练程序，用于批量创建用户 Profile 和行为模型。默认该训练系统每月更新一次模型，模型粒度为 1 分钟。

Eagle 内置的机器学习算法的基本思想如下。

1) 核密度估计算法（Density Estimation）

该算法的基本思想是，根据检测的训练样本数据，针对每个用户计算出对应的概率密度分布函数。首先将训练数据集的每个特征均值标准化，标准化可以使所有数据集转换为相同尺度。然后，在随机变量概率分布估计中，采用高斯分布式函数来计算概率密度。假设任意特征彼此相互独立，

那么最终的高斯概率密度就可以通过分解各个特征的概率密度而计算得到。在线实时检测阶段，可以首先计算出每个用户实时行为的概率。如果用户出现当前行为的可能性低于某个临界值，则标识为异常警告，而这个临界值完全由离线训练程序通过称为“马修斯相关系数”（Mathews Correlation Coefficient）的方法计算得出。

2) 特征值分解算法 (Eigen-Value Decomposition)

在该算法中，生成用户 Profile 的主要目的是从中发现有价值的用户行为模式。为了实现这个目的，可以考虑对特征依次进行组合，然后观察它们相互之间是如何影响的。当数据集非常大时，正如通常我们所遇到的场景，由于正常模式的数量非常多，以至于特征集的异常模式很容易被忽视。由于正常的行为模式通常处于非常低维的子空间内，因此，可以通过降低数据集的维度来更好地理解用户真正的行为模式。该方法同样可以对训练数据集进行降噪。根据对大量用户特征数据方差的运算，通常在用例场景中选取方差为 95% 的用户特征数据作为基准，则可以得到方差为 95% 的主成分的数量为 k ，因此将前 k 个主成分视为用户的正常子空间，而剩下的 $(n-k)$ 个主成分则被视为异常子空间。

在进行实时异常检测时，如果用户行为模式位于正常子空间附近，则认为该行为正常；如果用户行为模式位于异常子空间附近，则会立即报警，因为用户行为通常应该位于正常子空间内。至于用户当前行为是接近正常还是异常子空间，则采用欧氏距离法（Euclidian Distance Method）进行计算。

4. Eagle 服务

(1) 策略管理器：Eagle 策略管理器提供交互友好的用户界面和 REST API 供用户轻松地定义和管理策略，一切只需单击几下鼠标而已。Eagle 的用户界面使得策略的管理、敏感元数据的标识和导入、HDFS 或 Hive 的资源浏览及预警仪表等功能都非常易于使用。

Eagle 策略引擎默认支持 WSO2 Siddhi CEP 引擎和机器学习引擎。以下是几个基于 Siddhi CEP 的策略示例。

- 单一事件执行策略（用户访问 Hive 中的敏感数据列）：

```
from hiveAccessLogStream[sensitivityType=='PHONE_NUMBER'] select * insert into
outputStream;
```

- 基于窗口的策略（用户在 10 分钟内访问目录/tmp/private 多于 5 次）：

```
hdfsAuditLogEventStream[(src == '/tmp/private')]#window.externalTime(timestamp,
10 min) select user, count(timestamp) as aggValue group by user having aggValue >=
5 insert into outputStream;
```

(2) 查询服务 (Query Service)：Eagle 提供类 SQL 的 REST API 用来实现针对海量数据集的综合计算、查询和分析，支持如过滤、聚合、直方运算、排序、top、算术表达式及分页等功能。Eagle 优先支持 HBase 作为其默认的数据存储，同时也支持基于 JDBC 的关系型数据库。特别是当选择 HBase 作为数据存储时，Eagle 便原生拥有了 HBase 存储和查询海量监控数据的能力。Eagle 查询框架会将用户提供的类 SQL 查询语法最终编译成 HBase 原生的 Filter 对象，并支持通过 HBase

Coprocessor 进一步提升响应速度。

```
query=AlertDefinitionService[@dataSource="hiveQueryLog"]{@policyDef}&  
pageSize=100000
```

5.7 小结

从业界的发展和趋势来看，可以得出以下结论：

- 从架构角度来看，追求架构简单和维护简单的动力一直存在，所以会不断地有系统探索流处理和批处理的融合。相信随着技术的发展，未来会有一个很好的融合。
- 一般来说，随着时间的推移，数据的价值在降低，流系统使用的比例越来越高，流未来会成为默认应用。
- 当前基于流的分析基本处于空白，离线的分析算法未来会复制到流中。未来基于流的分析是工业界研究的重点。

第 6 章

交互式分析

定义：基于历史数据的交互式查询（Interactive Query），通常的时间跨度在数十秒到数分钟之间。

6.1 交互式分析的概念

在数据仓库领域有一个概念叫 Adhoc Query，中文一般译为“即席查询”。即席查询是指用户在使用系统时，根据自己当时的需求定义的查询。在大数据领域，扩展到 Interactive Query（交互式查询）是最常见的一种，通常用于客户投诉处理、实时数据分析、在线查询等。因为是查询应用，所以通常具有以下特点：

- 时延低（数据获取在数十秒到数分钟之间，可以查询到的数据近实时）。
- 查询条件复杂（多个维度，且维度不固定）。
- 查询范围大（通常查询表记录在几十亿条级别）。
- 返回结果数小（几十条甚至几千条）。
- 并发数要求高（几百、上千条同时并发）。
- 需要支持 SQL 等接口。

传统上，常常使用数据仓库来承担 Adhoc Query 的责任。为了提升查询体验、降低时延，数据库专家想了很多办法优化数据库，如常见的数据库索引、Sybase IQ 的列式存储等。

建立索引的思路是通过索引减少数据扫描，更适合传统的 TP 场景，如并发要求高、获取数据少的场景。列式存储则根据查询列的选择性比较强的特性来减少数据的读取，是一个不错的思路。

当数据库本身无法承担时，大家又想到使用内存缓存或 Cube 来承担这一任务。内存缓存是利用内存的速度，将数据提前缓存到内存中，提高缓存的命中率。Cube 的思路是将数据按所有维度预聚合好，数据仓库通过创建索引来应对多维度的复杂查询。

传统的一些做法在大数据时代仍会得到延续，但也存在明显的缺点，如扩展性不强、索引创建成本高、索引易失效等，需要一些并行处理技术来应对数据急剧增大的趋势。

本章讨论 MPP DB、SQL on Hadoop 及 Google 的系统，一起来看一下业界的解决思路。

6.2 MPP DB 技术

随着数据量的增大，传统数据库如 Oracle、MySQL、PostgreSQL 等单实例模式将无法支撑大量数据的处理，数据仓库采用分布式技术成为自然的选择。

6.2.1 MPP 的概念^①

在讨论 MPP DB 之前，我们先把 MPP 本身的概念搞清楚。MPP 是系统架构角度的一种服务器分类方法。

从系统架构来看，目前的商用服务器大体可以分为三类，即对称多处理器结构（Symmetric Multi-Processor, SMP）、非一致存储访问结构（Non-Uniform Memory Access, NUMA），以及海量并行处理结构（Massive Parallel Processing, MPP）。它们的特征分别描述如下。

1. SMP（Symmetric Multi-Processor）

所谓对称多处理器结构，是指服务器中的多个 CPU 对称工作，无主次或从属关系。各 CPU 共享相同的物理内存，每个 CPU 访问内存中的任何地址所需时间是相同的，因此 SMP 也被称为一致存储器访问结构（Uniform Memory Access, UMA）。对 SMP 服务器进行扩展的方式包括增加内存、使用更快的 CPU、增加 CPU、扩充 I/O（槽口数与总线数）及添加更多的外部设备（通常是磁盘存储）。

SMP 服务器的主要特征是共享，系统中的所有资源（如 CPU、内存、I/O 等）都是共享的。也正是由于这种特征，导致了 SMP 服务器的主要问题，即它的扩展能力非常有限。对于 SMP 服务器而言，每个共享的环节都可能造成 SMP 服务器扩展时的瓶颈，而最受限制的则是内存。由于每个 CPU 必须通过相同的内存总线访问相同的内存资源，因此，随着 CPU 数量的增加，内存访问冲突将迅速增加，最终造成 CPU 资源的浪费，使 CPU 性能的有效性大大降低。实验证明，SMP 服务器 CPU 利用率最好的情况是 2~4 个 CPU，如图 6.1 所示。

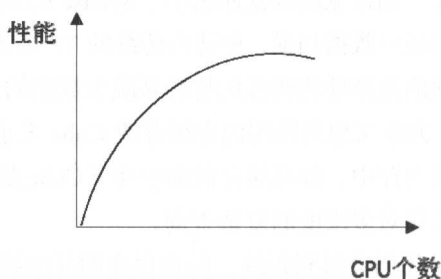


图 6.1

^① 参考知识：服务器三大体系 SMP、NUMA、MPP 介绍，<http://server.51cto.com/sCollege-198840.htm>。

2. NUMA (Non-Uniform Memory Access)

由于 SMP 在扩展能力上的限制,人们开始探究如何进行有效的扩展从而构建大型系统的技术, NUMA 就是这种努力下的结果之一。利用 NUMA 技术,可以把几十个 CPU (甚至上百个 CPU) 组合在一台服务器内。其 CPU 模块结构如图 6.2 所示。

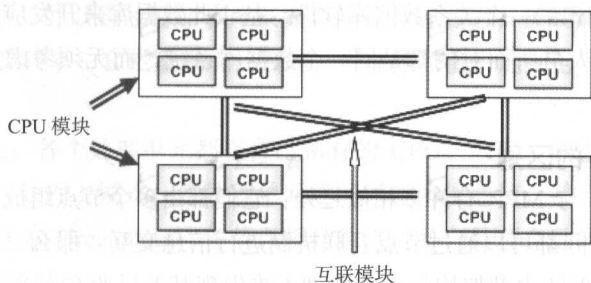


图 6.2

NUMA 服务器的基本特征是拥有多个 CPU 模块,每个 CPU 模块由多个 CPU (如 4 个)组成,并且具有独立的本地内存、I/O 槽口等。由于其节点之间可以通过互联模块 (如称为 Crossbar Switch) 进行连接和信息交互,因此,每个 CPU 可以访问整个系统的内存 (这是 NUMA 系统与 MPP 系统的重要区别)。显然,访问本地内存的速度将远远高于访问异地内存 (系统内其他节点的内存) 的速度,这也是非一致存储访问 NUMA 的由来。由于这个特点,为了更好地发挥系统性能,开发应用程序时需要尽量减少不同 CPU 模块之间的信息交互。利用 NUMA 技术,可以较好地解决原来 SMP 系统的扩展问题,在一台物理服务器内可以支持上百个 CPU。比较典型的 NUMA 服务器包括惠普的 Superdome、SUN15K、IBMp690 等。

但 NUMA 技术同样有一定的缺陷,由于访问异地内存的时延远远超过访问本地内存,因此,当 CPU 数量增加时,系统性能无法线性增加。如惠普公司发布 Superdome 服务器时,曾公布了它与惠普其他 UNIX 服务器的相对性能值,结果发现,64 路 CPU 的 Superdome 服务器 (NUMA 结构) 的相对性能值是 20,而 8 路 N4000 (共享的 SMP 结构) 的相对性能值是 6.3。从这个结果可以看出,8 倍数量的 CPU 换来的只是 3 倍性能的提升。

3. MPP (Massive Parallel Processing)

和 NUMA 不同, MPP 提供了另外一种进行系统扩展的方式,它由多台 SMP 服务器通过一定的节点互连网络进行连接,协同工作,完成相同的任务,从用户角度来看是一个服务器系统。其基本特征是由多台 SMP 服务器 (每台 SMP 服务器称为节点) 通过节点互连网络连接而成,每个节点只访问自己的本地资源 (内存、存储等),是一种完全无共享 (Share Nothing) 结构,因而扩展能力最强,理论上可以无限扩展,目前的技术可以实现 512 个节点互连,包含数千个 CPU。目前业界对节点互连网络暂无标准,如 NCR 的 Bynet、IBM 的 SPSwitch,它们都采用了不同的内部实现机制。但节点互连网络仅供 MPP 服务器内部使用,对用户而言是透明的。

在 MPP 系统中,每个 SMP 节点也可以运行自己的操作系统、数据库等。但和 NUMA 不同的是,

它不存在异地内存访问的问题。换言之，每个节点内的 CPU 不能访问另一个节点的内存。节点之间的信息交互是通过节点互联网络实现的，这个过程一般称为数据重分配（Data Redistribution）。

但是 MPP 服务器需要一种复杂的机制来调度和平衡各个节点的负载和并行处理过程。目前，一些基于 MPP 技术的服务器往往通过系统级软件（如数据库）来屏蔽这种复杂性。举例来说，NCR 的 Teradata 就是基于 MPP 技术的一个关系数据库软件，基于此数据库来开发应用时，不管后台服务器由多少个节点组成，开发人员所面对的都是同一个数据库系统，而无须考虑如何调度其中某几个节点的负载。

4. NUMA 与 MPP 的区别

从架构来看，NUMA 与 MPP 有许多相似之处：它们都由多个节点组成；每个节点都有自己的 CPU、内存、I/O；节点之间都可以通过节点互联机制进行信息交互。那么二者的区别在哪里？通过分析 NUMA 和 MPP 服务器的内部架构与工作原理不难发现其差异所在。

首先是节点互联机制不同。NUMA 的节点互联机制是在同一台物理服务器内部实现的，当某个 CPU 需要进行异地内存访问时，它必须等待，这也是 NUMA 服务器无法实现 CPU 增加时性能线性扩展的主要原因。而 MPP 的节点互联机制是在不同的 SMP 服务器外部通过 I/O 实现的，每个节点只访问本地内存和存储，节点之间的信息交互与节点本身的处理是并行进行的。因此，MPP 在增加节点时，其性能基本上可以实现线性扩展。

其次是内存访问机制不同。在 NUMA 服务器内部，任何一个 CPU 都可以访问整个系统的内存，但异地内存访问的性能远远低于本地内存访问，因此，在开发应用程序时应该尽量避免异地内存访问。而在 MPP 服务器中，每个节点只访问本地内存，不存在异地内存访问的问题。

5. 数据仓库的选择

哪种服务器更加适应数据仓库环境？这需要从数据仓库环境本身的负载特征入手。众所周知，典型的数据仓库环境具有大量复杂的数据处理和综合分析，要求系统具有很高的 I/O 处理能力，并且存储系统需要提供足够的 I/O 带宽与之匹配。而一个典型的 OLTP 系统则以联机事务处理为主，每次交易所涉及的数据不多，要求系统具有很高的事务处理能力，能够在单位时间里处理尽量多的交易。显然，这两种应用环境的负载特征完全不同。

从 NUMA 架构来看，它可以在一台物理服务器内集成多个 CPU，使系统具有较高的事务处理能力，但由于异地内存访问时延远长于本地内存访问，因此需要尽量减少不同 CPU 模块之间的数据交互。显然，NUMA 架构更适用于 OLTP 事务处理环境，当用于数据仓库环境时，由于大量复杂的数据处理必然导致大量的数据交互，将使 CPU 的利用率大大降低。

相对而言，MPP 服务器架构的并行处理能力更优越，更适合复杂的数据综合分析与处理环境。当然，它需要借助支持 MPP 技术的关系数据库系统来屏蔽节点之间负载平衡与调度的复杂性。另外，这种并行处理能力也与节点互联网络有很大的关系。显然，适应数据仓库环境的 MPP 服务器，其节点互联网络的 I/O 性能应该非常突出，这样才能充分发挥整个系统的性能。

6. MPP 数据仓库架构分类

前面讲到 MPP 架构非常复杂,通常用到数据库系统来屏蔽节点间的负载平衡和调度的复杂性。在数据库架构设计中,又有多种架构,主要分为 Share Disk 和 Share Nothing。

(1) Share Disk: 各个处理单元使用自己的私有 CPU 和 Memory,共享磁盘系统。典型代表是 OracleRac,它共享数据,可以通过增加节点来提高并行处理能力,扩展能力较好。处理节点采用的是 MPP 架构,但是需要共享一套磁盘系统,因此,当存储器接口达到饱和的时候,增加节点并不能获得更高的性能。

(2) Share Nothing: 各个处理单元都有自己私有的 CPU、内存、硬盘等,不存在共享资源,类似于 MPP (大规模并行处理) 模式,各处理单元之间通过协议通信,并行处理和扩展能力更好。典型代表是 DB2 DPF 版本和 Greenplum,各节点相互独立,各自处理自己的数据,处理后的结果可能向上层汇总或在节点间流转。

我们常说的 Sharding 其实就是 Share Nothing 架构,它把某张表从物理存储上水平分割,并分配给多台服务器(或多个实例),每台服务器可以独立工作,具备共同的 Schema,如 MySQL Proxy 和 Google 的各种架构,只需增加服务器数量就可以增加处理能力和容量。

Share Nothing 因为数据尤其是元数据存储在不同的服务器上,所以对各台服务器间的元数据同步及故障恢复来说是一个灾难。相对而言,Share Disk 不存在同步问题,计算节点故障后简单复位就可以恢复工作,但是存在共享存储导致的存储瓶颈问题。

6.2.2 典型的 MPP 数据库

1. Greenplum 架构^①

最早采用 MPP 架构的是 Teradata 数据库,整体上采用 Share Nothing 架构进行组织。Teradata 定位于大型数据仓库系统,定位较高,软硬件(包括协议)都是自己私有的,以一体机的形式销售,广泛应用于金融、证券、电信等行业。

Greenplum 数据库在开源的 PostgreSQL 的基础上采用了 MPP 架构,做出了性能非常强大的关系型分布式数据仓库。为了兼容 Hadoop 生态,又推出了 HAWQ,分析引擎保留了 Greenplum 的高性能引擎,下层存储不再采用本地硬盘而改用 HDFS,规避本地硬盘可靠性差的问题,同时融入 Hadoop 生态。

目前 EMC 已收购 Greenplum,和 VMware 一起成立一家新的公司 Pivotal^②来运营 Greenplum&HAWQ 等。

1) Share Nothing 架构

Greenplum 采用 Share Nothing 架构(MPP),主机、操作系统、内存、存储都是自我控制的,不存在共享。该架构主要由 Master Host(主节点)、Segment Host(工作节点)、Interconnect(内部通信)三大部分组成,如图 6.3 所示。

^① Greenplum 数据库引擎探究, <http://www.cnblogs.com/daduxiong/archive/2010/10/13/1850411.html>。

^② Pivotal 公司网址: <http://pivotal.io/>。

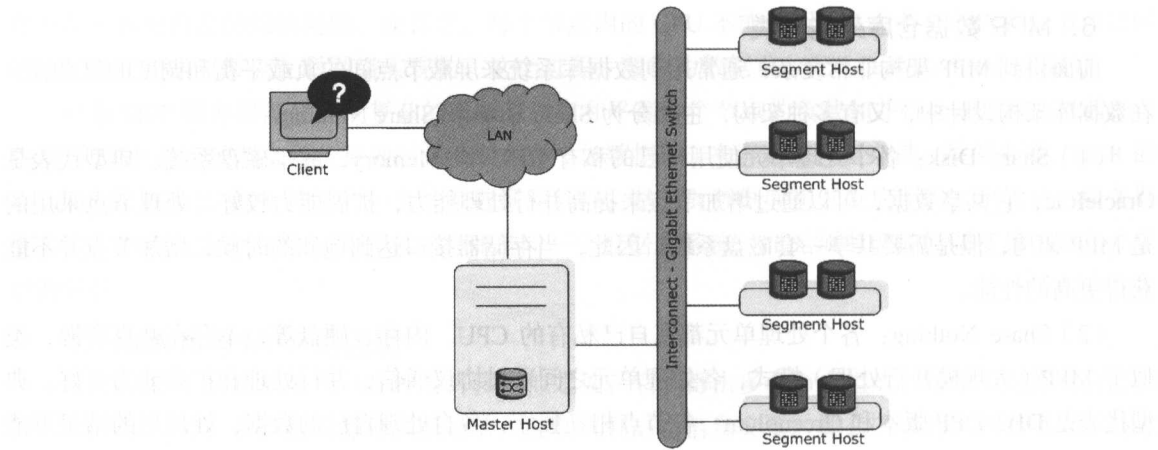


图 6.3

了解了 Greenplum 的架构,理解其工作流程也就相对简单了。因为 Greenplum 采用了 MPP 架构,其主要优点是大规模的并行处理能力,所以应该把主要精力放在大规模存储与并行处理两个方面。

① 大规模存储

Greenplum 数据库通过将数据分布到多个节点上来实现规模数据的存储。数据库的瓶颈经常发生在 I/O 方面,数据库的诸多性能问题最终总能归咎到 I/O 身上,久而久之,I/O 瓶颈成了数据库性能的永恒话题。

Greenplum 采用分而治之的方法,将数据规律地分布到节点上,充分利用 Segment 主机的 I/O 能力,以此让系统达到最大的 I/O 能力(主要是带宽),如图 6.4 所示。

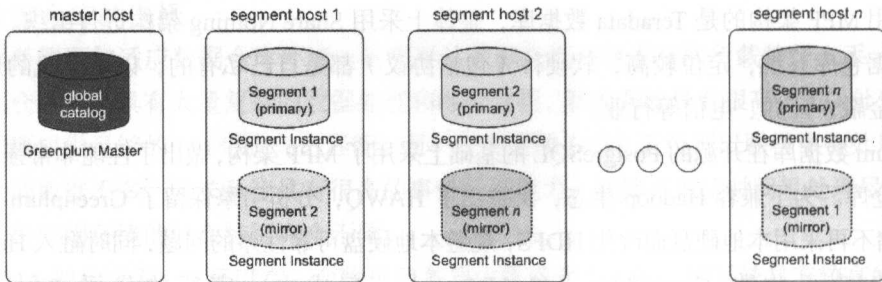


图 6.4

在 Greenplum 中,每张表都是分布在所有节点上的。Master Host 首先通过对表的某列或多列进行 Hash 运算,然后根据 Hash 结果将表的数据分布到 Segment Host 中。在整个过程中,Master Host 中不存放任何用户数据,只是对客户端进行访问控制和存储表分布逻辑的元数据。

② 并行处理

Greenplum 的并行处理主要体现在外部表并行装载、并行备份恢复与并行查询处理三个方面。

数据仓库的主要精力一般集中在数据的装载和查询上。数据的并行装载主要采用外部表或者 Web 表方式,通常情况下通过 gpfdist 程序来实现,如图 6.5 所示。

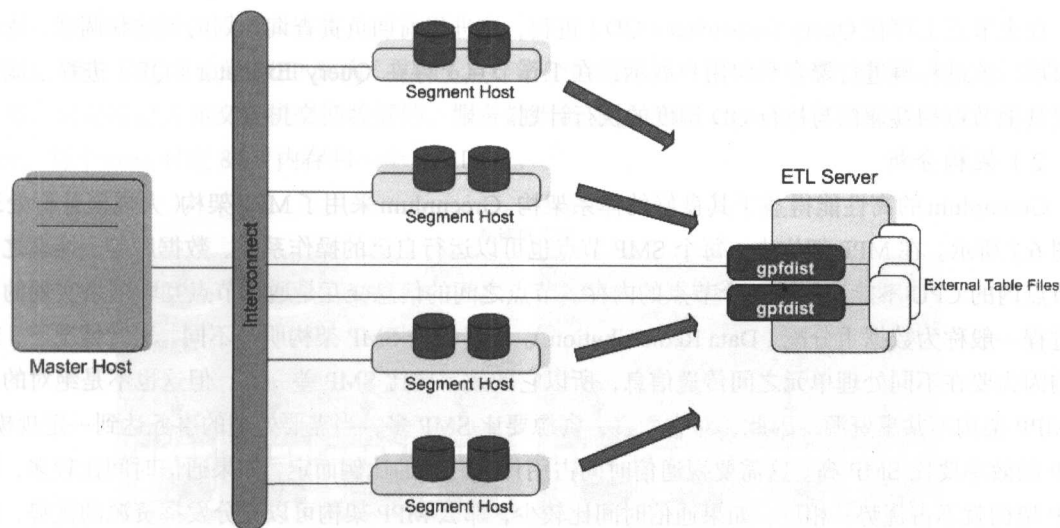


图 6.5

gpfdist 程序能够以 370MB/s 的速度装载 TEXT 格式的文件，以 200MB/s 的速度装载 CSV 格式的文件。在 ETL 带宽为 1GB 的情况下，可以同时运行 3 个 gpfdist 程序装载 TEXT 格式的文件，或者同时运行 5 个 gpfdist 程序装载 CSV 格式的文件。例如，在图 6.5 中采用两个 gpfdist 程序进行数据装载。可以根据实际环境，通过配置 postgresql.conf 参数文件来优化装载性能。

查询性能的强弱往往由查询优化器的水平来决定，Greenplum 主节点负责解析 SQL 与生成执行计划。Greenplum 的执行计划生成同样采用基于成本的方式，由于数据库是由诸多 Segment 实例组成的，所以在选择执行计划时，主节点还要综合考虑节点间传送数据的代价。

其工作原理如图 6.6 所示。

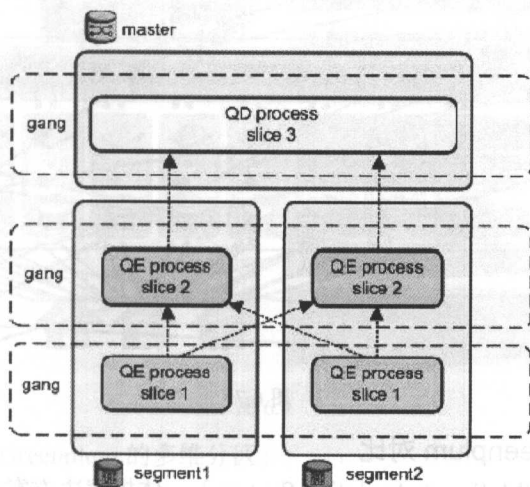


图 6.6

在主节点上存在 Query Dispatcher (QD) 进程, 该进程前期负责查询计划的创建和调度, 待返回结果后, 该进程再进行聚合和向用户展示; 在工作节点上存在 Query Executor (QE) 进程, 该进程负责其他节点相互通信与执行 QD 调度的执行计划。

2) 架构分析

Greenplum 的高性能得益于其良好的体系架构。Greenplum 采用了 MPP 架构(大规模并行处理), 如图 6.7 所示。在 MPP 架构中, 每个 SMP 节点也可以运行自己的操作系统、数据库等。换言之, 每个节点内的 CPU 不能访问另一个节点的内存。节点之间的信息交互是通过节点互连网络实现的, 这个过程一般称为数据重分配 (Data Redistribution)。与传统的 SMP 架构明显不同, 通常情况下, MPP 架构因为要在不同处理单元之间传送信息, 所以它的效率要比 SMP 差一点, 但这也不是绝对的, 因为 MPP 架构不共享资源, 因此, 对它而言, 资源要比 SMP 多, 当需要处理的事务达到一定规模时, MPP 的效率要比 SMP 高。这需要视通信时间占用计算时间的比例而定, 如果通信时间比较多, 那么 MPP 架构就不占优势; 相反, 如果通信时间比较少, 那么 MPP 架构可以充分发挥资源的优势, 达到高效率。在当前使用的 OTLP 程序中, 用户访问一个中心数据库, 如果采用 SMP 架构, 其效率要比采用 MPP 架构高得多。而 MPP 架构在决策支持和数据挖掘方面有明显的优势, 可以这样说, 如果操作相互之间没有什么关系, 处理单元之间需要进行的通信比较少, 那么采用 MPP 架构较好; 相反就不合适了。

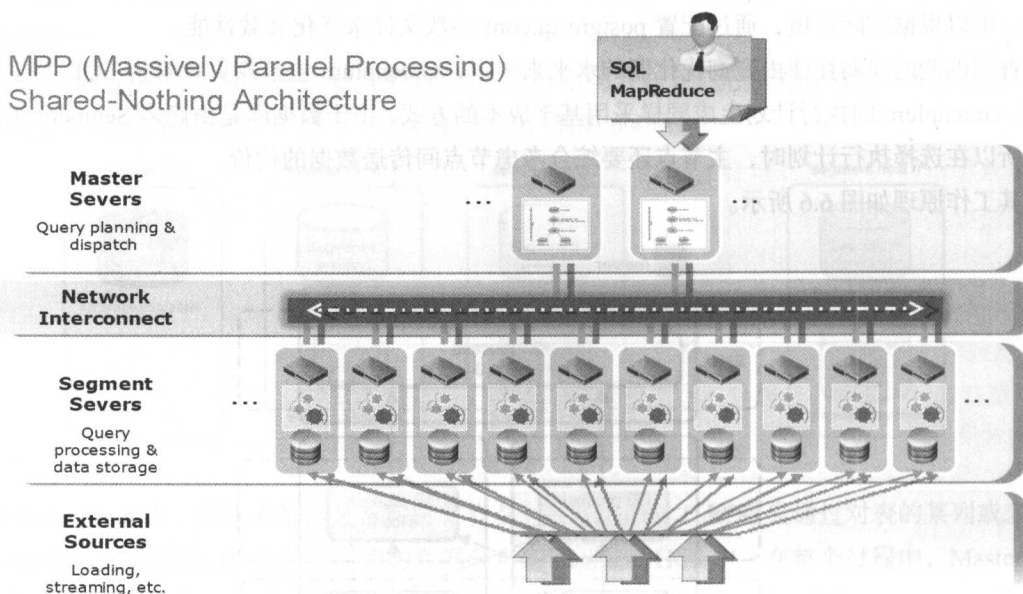


图 6.7

2. DB2 DPF 和 Greenplum 对比

IBM 推出的 ISAS (IBM Smart Analytics System) 一体机解决方案里面装载的是 DB2 的 DPF (Database Partitioning Feature) 版本, 采用的也是 MPP 架构, 下面就来看看二者的区别。

1) 架构相似

DB2 DPF 采用的也是 MPP 架构 (见图 6.8), 每个数据库都有独立的日志、引擎、锁、缓存管理。服务器之间是通过万兆交换机交换数据的。服务器内部通过 share_memory 实现互相访问。服务器为 16core, 每个 core 对应 8GB 内存和一个 RAID 组。

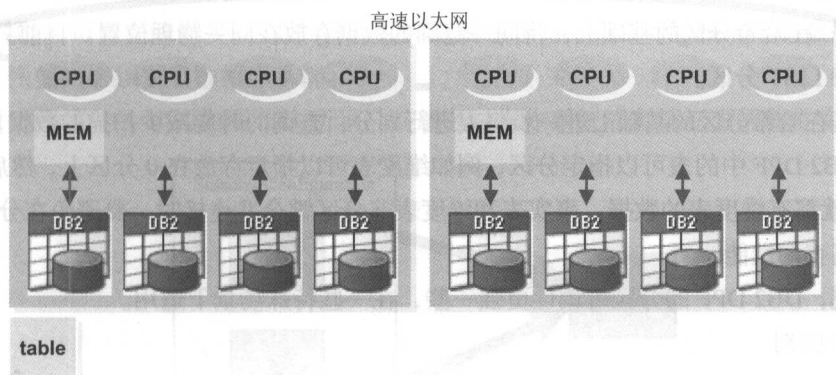


图 6.8

2) 分区技术

在 MPP 架构中解决了各个节点的并行处理问题。Greenplum 和 DB2 DPF 都采取了同样的思路——表分区，就是将一张完整的表，通过 Hash 算法，尽量均衡地分布在不同的节点上，如图 6.9 所示。

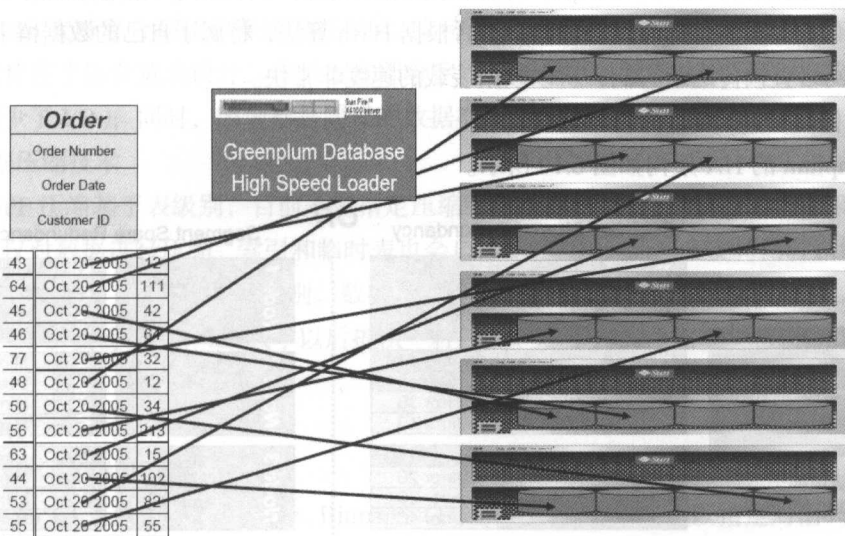


图 6.9

下面讲解 DB2 DPF 和 Greenplum 的多维分区。

① 哈希分区

DB2 DPF 中的分区键必须指定，没有类似 Greenplum 中随机分区概念。如果没有一个合适的

列作为分区键，则可以通过给表新增一个自动生成列，列中填充随机数据，然后以这个自动生成列作为分区键。支持多列作为分区键，数据库自动通过这些列计算出 Hash 值，然后决定分区位置。在指定分区键后，会生成 Hash map，如果数据不均匀，则可以通过调整 Hash map 微调数据分布。目前在 Greenplum 中没有看到这个功能，应该说 DB2 DPF 的功能更强大一些。

② 表分区

DB2 DPF 在哈希分区的基础上，将同一范围的数据存放在同一物理位置。目前只支持 RANGE 分区，不支持 LIST 分区。

表分区是在哈希分区的基础上进一步将表进行划分，查询的时候减少扫描的数据量，减少 I/O。

其中，DB2 DPF 中的表可以指定分区，例如维度表可以指定存放在 0 分区上，然后通过表复制，32 个分区上就都有维度表的数据。事实表和维度表非分区键合并连接时，要避免在分区之间发送数据，从而提高查询性能。

总的来说，DB2 DPF 的分区可调性更强一些，在一些特殊场景下适用。

3) 细节区别

两种都是 MPP 架构的数据，设计思路类似，但在一些细节上还是有区别的。

① 数据装载

Greenplum Master 节点只承担少量的控制功能，以及和客户端的交互，完全不承担任何计算。DB2 DPF 装载必须由 admin 节点来完成，通过 admin 节点上的多进程对数据进行分发。装载需要消耗一定的性能。与之相反的是，Greenplum 在进行数据装载时，不是我们一般想象的存在一个中心数据分发节点，而是所有节点同时读取数据，然后根据 Hash 算法，将属于自己的数据留下，将其他节点的数据通过网络直接传送给它们，所以数据装载的速度非常快。

② HA 架构

(1) Greenplum 的 HA 架构如图 6.10 所示。

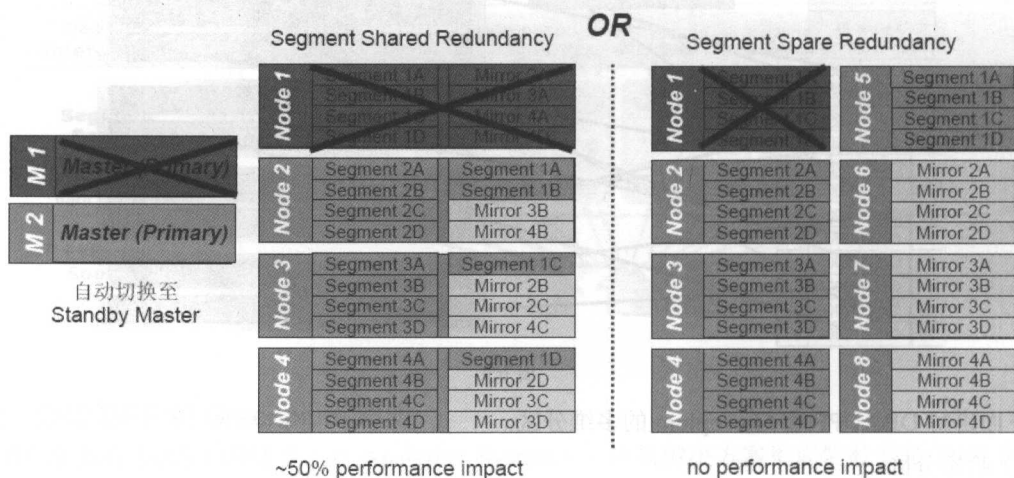


图 6.10

Master 是通过单独的 Host 进行冗余备份的。其最大的特点是 Segment 通过 mirror（镜像）来实现冗余。Segment 镜像和 Segment 保存在不同的 Host 上，Master 如果连接不上，就标记为 invalid 状态；下次连接上了，就标记为 valid 状态。

系统如果没有配置镜像，那么 Master 检测 invalid 的 Segment 时，就会关闭数据库来保证数据不出错。如果系统配置了镜像，那么系统在 read-only 和 continue 模式下的处理方式不同，前者不允许 DDL 和 DML 操作，可以在线恢复；后者的操作必须限制在非 invalid 的 Segment 上的数据，而且 invalid 的 Segment 在恢复时必须重启数据库系统。

(2) DB2 DPF 的 HA 架构如图 6.11 所示。一般通过操作系统或者第三方的软件实现 HA。

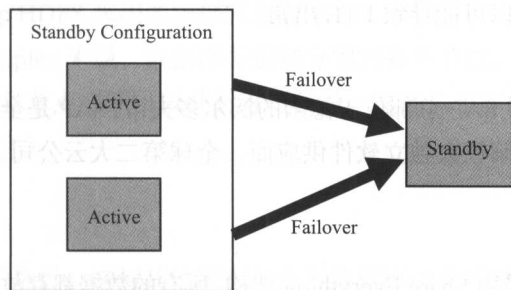


图 6.11

③ 数据存储

DB2 DPF 只支持行存储。

Greenplum 支持混合存储和普通的行存储，支持读/写及列存储只读，不支持 update、delete 操作。列存储的优势在于适合宽表设计，在只查询部分字段的情况下，效率高于行模式（因为只需读出相应的列，减少了 I/O）；同时，因为都是同样的数据类型，所以更容易压缩。

④ 数据压缩技术

DB2 DPF 压缩基于表级别，目前不能指定压缩级别。提供类似 WinZip 的压缩级别。ISAS 压缩的特点是不仅对数据进行压缩，索引和临时表也会自动压缩。Greenplum 支持两种压缩算法：一种是 ZLIB，压缩比较高，提供 1~9 个级别，数字越大，压缩比越高；另一种是 QUICKLZ，其压缩比较小，相应的 CPU 负荷较低。4.2 版本以后提供一种新的、基于列的压缩算法 RLE，提供基于列级别的压缩。

⑤ 索引技术

DB2 DPF 只支持 B+索引。

Greenplum 支持三种索引：B-Tree、Bitmap 和 Hash。Greenplum 还可以指定对语句创建部分索引。索引影响 insert、update 操作，创建的时候消耗 CPU。

创建索引须遵守以下原则：

- (1) 不要对经常变更的列创建索引。只有在全表扫描性能不好时才需要创建索引。
- (2) 不要创建重复的索引并给索引命名。

(3) 低基数的列使用 Bitmap 索引；单列查询使用 B-Tree 索引。

(4) 加载数据的时候先删除索引，加载完成后再重新创建索引。

(5) 扫描一张大表的子集时，使用部分索引。

(6) 重新创建索引执行 Analyze 操作。

⑥ 在线扩容

二者都支持在线扩容，扩容时，表数据需要重新分布。在进行表重分布时，是一张一张表进行的。正在进行数据重分布的表不能加载数据。这点类似于 Greenplum，Greenplum 在进行数据重分布时，正在重分布的表不能读/写。另外，Greenplum 会自动去掉唯一性限制，所以在进行表重分布时，遇到重复的行不会报错，所以可能导致 ETL 出错。

3. Sybase IQ

Sybase IQ 数据的母公司 SAP 总部位于德国的沃尔多夫市，SAP 是全球最大的企业管理和协同化商务解决方案供应商、世界第三大独立软件供应商、全球第二大云公司。Sybase IQ 是一款非常独特的 MPP 数据库。

1) 架构演变^①

IQ 在 16.0 版本以前都采用 Share Everything 架构，所有的数据都存放在一个共享的 SAN 存储中，和 Oracle RAC 的架构类似。从 IQ 16.0 SP10 开始引入 Share Nothing Multiplex/MPP 的支持。“Share Nothing Multiplex”是一个在大数据环境下针对大规模并行处理（Massively Parallel Processing, MPP）的存储和处理架构。在这个存储架构下，主数据（Primary Data）存储在—组节点中的直插式存储（Direct-Attached Storage, DAS）设备集合中，而不是存储在一个共享存储区域网络设备中（SAN 存储设备中）。

Share Nothing Multiplex 的优势主要有以下几点。

① I/O 扩展能力提升

此前的 IQ Multiplex 采用 Share Disk 存储架构来存放数据，当节点增加、数据量不断增长时，共享存储成为瓶颈。Share Nothing Multiplex 采用分布式存储技术，极大地提升了 I/O 扩展能力。每个节点中的本地 DAS 存储设备能够实现比共享 SAN 存储设备更高的 I/O 性能。

IQ 16.0 SP10 引入的 Share Nothing Storage 类似于 Hadoop 的 HDFS，使用每个节点自己本地的存储存放用户数据的子集。

在 IQ Multiplex 中，可以同时使用 Share Nothing dbspaces 和 Share Disk dbspaces；在一个物理集群中，Share Nothing Multiplex 和 Share Disk Multiplex 可以同时存在。

② 提供更强的数据保护

每个节点中的 DAS 设备可以指定镜像文件，通过镜像技术能够提升数据的高可用性。这与 Hadoop 的 HDFS 也有一些类似，都是通过数据冗余技术来避免数据损坏、丢失。

^① 参考 <http://blog.chinaunix.net/uid-16765068-id-5132568.html>。

③ 存储设备管理变得相对简单

之前管理 IQ Multiplex 共享存储设备时, 需要管理裸设备。Share Disk 存储结构要求每个节点上的设备路径一定要相同, 并且指向一定要正确。当节点数量比较多或者共享裸设备数量比较多时, 会造成管理的负担, 并且容易出现错误。

IQ 16.0 SP10 的 Share Nothing Multiplex 采用了分布式存储架构, 在每个节点上可以使用裸设备, 也可以使用文件系统中的文件。

④ 数据库备份的变化

由于采用了节点本地的设备镜像技术, 因而 IQ Share Nothing Multiplex 的数据保护能力得到了进一步的提升 (类似于 Hadoop HDFS 采用的数据冗余技术)。

对于 Share Nothing Multiplex 来说, 数据库备份被分解到各个节点, 每个节点只负责备份自己拥有的本地设备上的数据, 并且节点之间可以并行进行备份。这降低了原先 Share Disk Multiplex 备份时对于备份设备的压力。

2) IQ 的独特优势

① 列存储

IQ 以列存储数据, 而不是行, 这与其他所有关系型数据库引擎广泛使用的存储方向相反。在其他关系型数据库内核中, 数据库的一张表的典型表示为一条数据库页链, 每个数据页中有一行或多行数据记录。在数据仓库应用中, 从查询性能的观点出发, 这种存储方式是所有可能的数据存储方式中最不可取的。在 IQ 中, 每张表是一组相互独立的页链的集合, 每条页链代表表中的一列。所以有 100 列的表将有 100 条相互独立的页链, 每一列都有一条页链与之相对应, 而不像其他数据库引擎, 一张表对应一条页链。列存储所固有的优越性在于: 大多数数据仓库应用的查询只关心表中所有列的一个很小的子集, 从而可以以很少的磁盘 I/O 得到查询结果。现在考虑这样一个例子, 假设我们要得到所有生日在 7 月份的客户的名字和电子邮件地址。

在一个典型的 OLTP 数据库引擎中, 查询优化器将根据返回行的百分比 (如 1/12, 在本例中, 假设各月的生日基本平均) 来决定是否值得在该列上使用索引。因此, 典型的数据库引擎对该查询可能会做全表的扫描。为了对扫描的成本进行估算, 假设每个客户的行记录为 3200 字节, 共有 1000 万条记录。因此, 表扫描必须读取 320 亿字节的数据。IQ 数据库引擎可以只读取查询所需的列。在本例中, 有三个相关的列: 全名、电子邮件地址和出生日期。假设全名为 25 字节, 电子邮件地址为 25 字节, 出生日期为 4 字节 (日期以二进制进行内部编码), 那么, IQ 只需读取 5400 万字节的数据, 大约减少了 59 倍。

② 数据压缩

传统的数据库引擎不能以一种通用的方式进行数据压缩, 主要是因为存在以下三个问题:

- 按行存储的数据存储方式不利于压缩。这是因为数据 (大多数为二进制数据) 在以这种方式存储时重复并不多。我们发现, 按行存储的数据, 最多有 510% 的压缩比例。
- 对于许多 2K 和 4K 的二进制数据的页来说, 为压缩和解压缩而增加的开销太大。

- 在 OLTP 环境中，大量读取和更新混杂在一起。每次更新都需要进行压缩操作，而读取只需进行解压缩操作，大多数数据压缩算法在压缩时比解压缩时慢 4 倍。这一开销将明显降低 OLTP 数据库引擎的事务处理效率，从而使得数据压缩的代价昂贵到几乎不能忍受。

在数据仓库应用中，数据压缩可以用小得多的代价换取更大的好处。其中包括减少对于存储量的要求、增大数据吞吐量，这相当于减少查询响应时间。

Sybase IQ 使用了数据压缩。这是由于数据按列存储时，相邻的字段值具有相同的数据类型，其二进制值的范围通常也要小得多，所以压缩更容易，压缩比更高。Sybase IQ 对列存储的数据通常能得到大于 50% 的压缩。更大的压缩比例，加上大页面 I/O，使得 Sybase IQ 在获得优良的查询性能的同时，减少了对存储空间的需求。

IQ 独特的技术特征所带来的一些关键的应用效果如下。

① 查询效果，瞬间响应

IQ 通过列存储、革命性的位图索引方法及智能的动态访问技术实现了更快的查询响应速度，比传统的数据库查询速度提高 10~1000 倍。这主要表现在如下几个方面：

(1) 减少磁盘 I/O。Sybase IQ 通过独特的列存储、索引与压缩技术，大大减少了查询中的磁盘 I/O 次数，其杰出的磁盘 I/O 效果带来了更快速的查询反应、更高的吞吐量和更低的成本。

(2) 并行列处理。IQ 支持列向量的并行处理，这样，在查询中，大量的列向量将被并行扫描，从而达到显著降低响应时间的目的。

(3) 智能优化。IQ 允许在每个列上建立多个索引，IQ 查询优化器可以在不同的使用情况下为查询选择不同的索引。

(4) 提高 Cache 命中率。大多数传统的关系型数据库执行决策支持类型的查询时会进行表扫描。表扫描会使 Cache 命中率降低。列存储方式使 Cache 命中率大大提高，查询响应速度加快。

② IQ 并行结构对多用户查询的性能影响

大多数传统数据库采用的并行表扫描方法在一个大型 SMP 上只有一个用户的情况下，效果是最好的，但在多用户查询环境中的性能会大打折扣。原因是现在的大多数 SMP 系统只能同时支持一至两张大型表的并行扫描，如果扫描数量增加，不是 CPU 资源不够，就是耗尽了 I/O 总线的带宽。在进行表扫描的同时也使数据库缓冲完全失效，因为大多数大型数据仓库应用的表扫描都远大于物理缓冲区的存储能力。IQ 独特的并行结构可以在大量的并发查询情况下提供优秀的查询性能。

③ 存储效果，节约存储成本

智能压缩技术与精巧的索引和列存储相结合，使得 IQ 比其他数据库引擎拥有更好的存储效果。这将获得更低的存储成本与更高的查询性能（因为系统仅需很少的磁盘 I/O 读取或写入任何给定的数据库块）。

在传统的数据库中，为提高查询性能所建的索引占用的磁盘空间往往需要比数据本身所需的磁盘空间多出 310 倍。而 Sybase IQ 存储数据所占用的磁盘空间通常只是原数据文件的 40%~60%，大大节约了存储成本。

④ 数据加载，更高效率

数据加载，包括向一个现有的数据仓库中增加数据，其加载效率大大高于传统的数据库。这是因为 IQ 的列存储、列并行处理与索引技术为快速批量数据加载提供了强大的技术保证。如果在数据加载时考虑索引的增量式加载，则大多数传统的关系型数据库在数据的增量式加载方面都存在严重的问题。这就是说，如果一次装入 100GB 的原始数据并在其上创建索引，再在此基础上增加 100MB 或几 GB 的数据会非常慢，以至于如果先删除所有的索引，再装入增加的数据，然后重新对整个数据库创建索引反而会快一些。Sybase IQ 在装入第二个 100MB 或 100GB 的数据时的速度几乎与装入第一批数据一样快，这就给最终用户带来了更大的灵活性。Sybase IQ 使得索引的开销大大降低，并提供了一个新的开发环境，在这样的开发环境中，索引的使用可以比过去任何时候更加充分。

6.2.3 MPP DB 调优实战

前面介绍了 Greenplum 及 Sybase IQ 等多种数据库，在实际使用中，要发挥数据库的最大效果，调优必不可少，下面介绍 Linux 系统及数据库的调优技术。

1. Linux 系统调优原理^①

性能调优是一项非常艰难的任务，它要求对硬件、操作系统和应用都有着相当深入的了解。如图 6.12 所示，服务器的性能受到众多因素的影响。

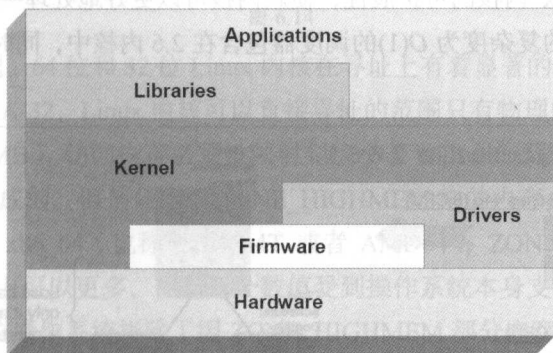


图 6.12

当面对一台使用单独 IDE 硬盘的、有 20 000 用户的数据库服务器时，即使我们使用数周时间去调整 I/O 子系统也是徒劳的，但通常一个新的驱动或者应用程序的一次更新（如 SQL 优化）就可以使这台服务器的性能得到明显的提升。正如前面提到的，不要忘记系统的性能是受多方面因素影响的。理解操作系统管理系统资源的方法，将有助于我们在面对问题时更好地判断应该对哪个子系统进行调整。

① <http://os.51cto.com/art/201303/384252.htm>。本文截取和修改自 IBM 的红皮书：<http://www.redbooks.ibm.com/redpapers/pdfs/redp3861.pdf>。

1) Linux 的 CPU 调度

任何计算机的基本功能都十分简单，那就是计算。为了实现计算的功能，就必须有一种方法去管理计算资源、处理器和计算任务（也称线程或者进程）。非常感谢 Ingo Molnar，它为 Linux 内核带来了复杂度为 $O(1)$ 的 CPU 调度器。区别于旧有的复杂度为 $O(n)$ 的调度器，新的调度器是动态的，可以支持负载均衡，并以恒定的速度进行操作。

新调度器的可扩展性非常好，无论是进程数量还是处理器数量，并且调度器本身的系统开销更小。新调度器的算法使用两个优先级队列：

- 活动运行队列。
- 过期运行队列。

调度器的一个重要目标是根据优先级权限有效地为进程分配 CPU 时间片，当分配完成后，它被列入 CPU 的活动运行队列中。除了 CPU 的活动运行队列外，还有一个过期运行队列。当活动运行队列中的一个任务用完自己的时间片后，它就被移动到过期运行队列中。在移动过程中，会对其时间片重新进行计算。如果活动运行队列中已经没有某个给定优先级的任务，那么指向活动运行队列和过期运行队列的指针就会交换，这样就可以让过期优先级列表变成活动优先级列表。通常交互式进程（相对于实时进程而言）都有一个较高的优先级，它占有更长的时间片，可以比低优先级的进程获得更多的计算时间，但通过调度器自身的调整并不会使低优先级的进程完全被忽略。新调度器的优势是显著地改变 Linux 内核的可扩展性，使新内核可以更好地处理一些由大量进程和大量处理器组成的企业级应用。新的复杂度为 $O(1)$ 的调度器包含在 2.6 内核中，同时也向下兼容 2.4 内核，如图 6.13 所示。

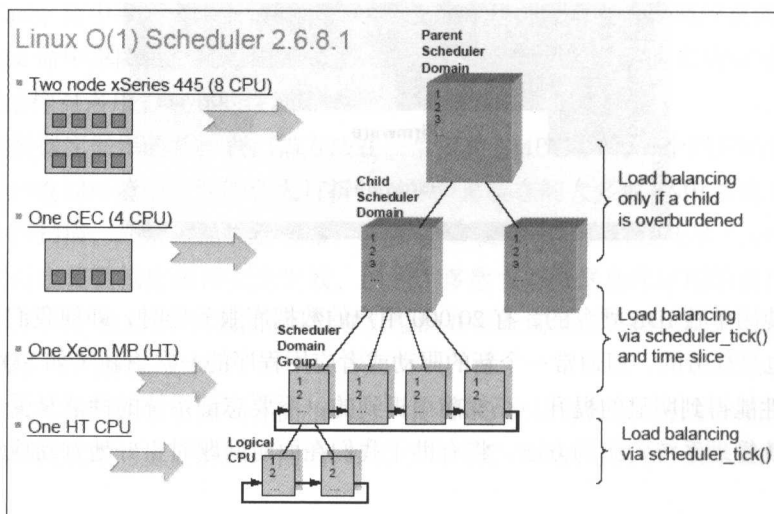


图 6.13

新调度器的另一个重要优势体现在对 NUMA 和 SMP 的支持上，如 Intel 的超线程技术。改进的 NUMA 支持保证了负载均衡不会发生在 CECs 或者 NUMA 节点之间，除非一个节点超出负载限度。

2) Linux 的内存架构

今天我们面对是选择 32 位操作系统还是 64 位操作系统的情况。对于企业级用户来说，它们之间最大的区别是 64 位操作系统可以支持大于 4GB 的内存寻址。从性能角度来讲，我们需要了解 32 位和 64 位操作系统都是如何进行物理内存和虚拟内存的映射的，如图 6.14 所示。

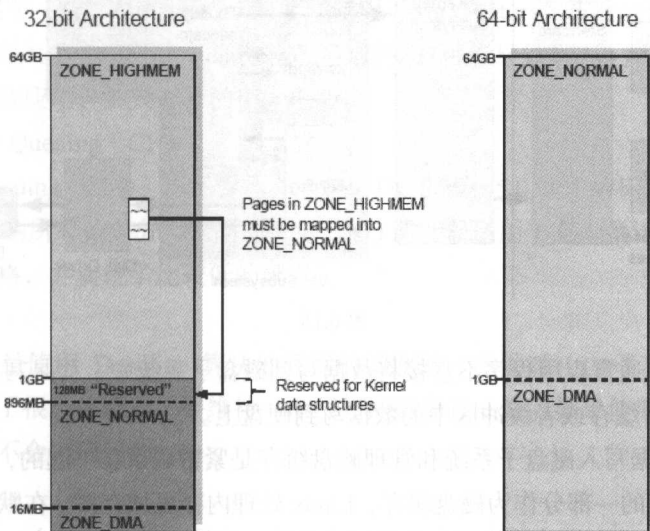


图 6.14

在图 6.14 中可以看到，64 位和 32 位 Linux 内核在寻址上有着显著的不同。

在 32 位架构中，如 IA-32，Linux 内核可以直接寻址的范围只有物理内存的第一个 GB（如果去掉保留部分，还剩下 896MB），访问内存必须被映射到这小于 1GB 的所谓 ZONE_NORMAL 空间中。这个操作是由应用程序完成的。但是分配在 ZONE_HIGHMEM 中的内存页将导致性能降低。

在 64 位架构中，如 x86_64（也称作 EM64T 或者 AMD64），ZONE_NORMAL 空间将扩展到 64GB 或者 128GB（实际上可以更多，但是这个数值受到操作系统本身支持内存容量的限制）。正如我们所看到的，使用 64 位操作系统排除了因 ZONE_HIGHMEM 部分内存对性能的影响的情况。

实际上，在 32 位架构下，由于上面所描述的内存寻址问题，对于大内存、高负载应用来说，会导致死机或严重缓慢等问题。虽然使用 hugemem 核心可以缓解，但采取 x86_64 架构是最佳的解决办法。

3) 虚拟内存管理

因为操作系统将内存都映射为虚拟内存，所以操作系统的物理内存结构对于用户和应用来说通常都是不可见的。如果想要理解 Linux 系统内存的调优，则必须了解 Linux 的虚拟内存机制。应用程序并不分配物理内存，而是向 Linux 内核请求一部分映射为虚拟内存的内存空间。如图 6.15 所示的虚拟内存并不一定是映射物理内存中的空间，如果应用程序有一个大容量的请求，那么也可能被映射到磁盘子系统的 Swap 空间中。

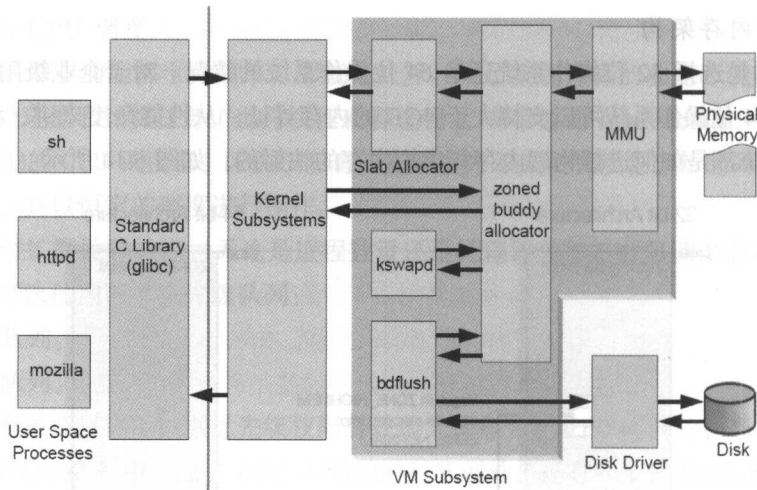


图 6.15

另外要提到的是，通常应用程序不直接将数据写到磁盘子系统中，而是写入缓存和缓冲区中。bdfush 守护进程定时将缓存或者缓冲区中的数据写到硬盘上。

Linux 内核处理数据写入磁盘子系统和管理的磁盘缓存是紧密联系在一起，相对于其他操作系统都是在内存中分配指定的一部分作为磁盘缓存，Linux 处理内存更加有效。在默认情况下，虚拟内存管理器分配所有可用内存空间作为磁盘缓存，这就是为什么有时我们观察一个配置有数 GB 内存的 Linux 系统可用内存只有 20MB 的原因。

同时 Linux 使用 Swap 空间的机制也是相当高效的，如图 6.15 所示的虚拟内存空间是由物理内存和磁盘子系统中的 Swap 空间共同组成的。如果虚拟内存管理器发现一个已经分配完成的内存分页已经长时间没有被调用，那么它将把这部分内存分页移到 Swap 空间中。我们经常会发现一些守护进程，如 getty，会随系统启动，但是很少会被应用到。这时为了释放昂贵的主内存资源，系统会将这部分内存分页移动到 Swap 空间中。上述就是 Linux 使用 Swap 空间的机制，当 Swap 分区使用超过 50% 时，并不意味着物理内存的使用已经达到瓶颈，Swap 空间只是 Linux 内核更好地使用系统资源的一种方法。

简单理解，Swap usage 只表示了 Linux 管理内存的有效性。对于识别内存瓶颈来说，Swap In/Out 才是比较有意义的依据。如果 Swap In/Out 的值长期保持在每秒 200~300 个页面，通常就表示系统可能存在内存的瓶颈。下面的示例是好的状态：

```
# vmstat
procs ---memory----- -swap- --io-- -system- --cpu--
r b swpd free buff cache si so bi bo in cs us sy id wa
1 0 5696 6904 28192 50496 0 0 88 117 61 29 11 8 80 1
```

4) 模块化的 I/O 调度器

Linux 2.6 内核为我们带来了很多新的特性，其中就包括新的 I/O 调度机制。Linux 2.4 内核使用一个单一的 I/O 调度器，而 2.6 内核则提供了 4 个可选的 I/O 调度器。因为 Linux 系统应用范围很

广，不同的应用对 I/O 设备和负载的要求都不相同，例如，一台笔记本电脑和一台有着 10 000 用户的数据库服务器对 I/O 的要求肯定有很大的区别。

① Anticipatory

Anticipatory I/O 调度器假设一个块设备只有一个物理的查找磁头（例如，一块单独的 SATA 硬盘）。正如调度器的名字一样，Anticipatory 调度器使用“Anticipatory”算法向硬盘写入一个比较大的数据流，以此来代替写入多个随机的小的数据流，这样就有可能导致写 I/O 操作的一些延迟。这个调度器适用于通常的一些应用，如大部分的个人电脑。

② Complete Fair Queuing (CFQ)

Complete Fair Queuing (CFQ) 调度器是 Red Flag DC Server 5.0 使用的标准算法。CFQ 调度器使用 QoS 策略为系统内的所有任务分配相同的带宽。CFQ 调度器适用于有大量计算进程的多用户系统。它试图避免进程被忽略，并实现了比较低的延迟。

③ Deadline

Deadline 调度器是使用 Deadline 算法的轮询的调度器，提供对 I/O 子系统接近实时的操作。Deadline 调度器提供了很小的延迟，并维持一个很好的磁盘吞吐量。如果使用 Deadline 算法，那么请确保进程资源分配不会出现问题。

④ NOOP

NOOP 调度器是一个简化的调度程序，它只执行最基本的合并与排序操作。其与桌面系统的关系不是很大，主要用在一些特殊的软件与硬件环境下。这些软件与硬件一般都拥有自己的调度机制，对内核支持的要求很小，很适合一些嵌入式系统环境。作为桌面用户，我们一般不会选择它。

5) 网络子系统

新的网络中断缓和 (NAPI) 给网络子系统带来了改变，提高了大流量网络的性能。Linux 内核在处理网络堆栈时，相比降低系统占用率和高吞吐量，更关注可靠性和低延迟。所以在某些情况下，Linux 建立一个防火墙或者文件、打印、数据库等企业级应用的性能可能会低于相同配置的 Windows 服务器。

在传统的处理网络封包的方式中，正如图 6.16 中的箭头所描述的，一个以太网封包到达网卡接口后，如果 MAC 地址相符合，则会被送到网卡的缓冲区中。然后网卡将封包移到操作系统内核的网络缓冲区中，并且对 CPU 发出一个硬中断，CPU 会处理这个封包到相应的网络堆栈中，可能是一个 TCP 端口或者 Apache 应用中。

这是一个处理网络封包的简单流程，但从中可以看到这种处理方式的缺点。每次有适合的网络封包到达网络接口时，都将对 CPU 发出一个硬中断信号，中断 CPU 正在处理的其他任务，导致切换动作和对 CPU 缓存的操作。在只有少量的网络封包到达网卡的情况下，这并不是问题，但是千兆网络和现代应用将带来每秒成千上万的网络数据，这就有可能对性能造成不良的影响。

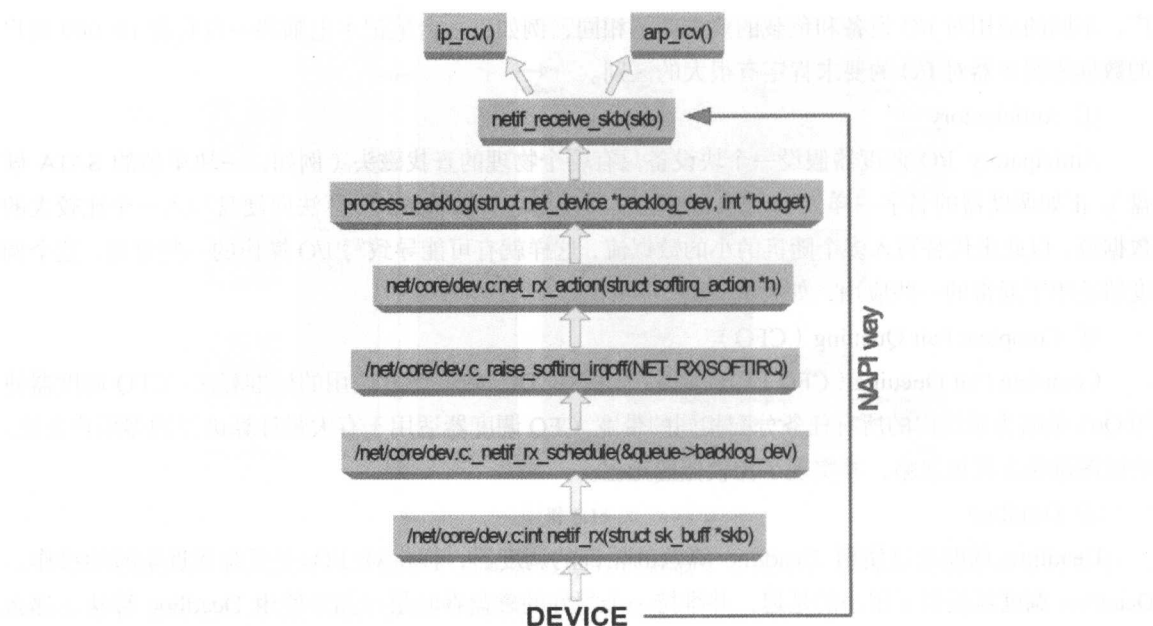


图 6.16

正因如此，NAPI 在处理网络通信的时候引入了计数机制。对于第一个封包，NAPI 以传统的方式进行处理；但是对于后面的封包，网卡引入了 POLL 的轮询机制：如果一个封包在网卡 DMA 环的缓存中，就不再为这个封包申请新的中断，直到最后一个封包被处理或者缓冲区被耗尽。这样就有效地减少了因为过多地中断 CPU 对系统性能的影响。同时，NAPI 通过创建可以被多处理器执行的软中断改善了系统的可扩展性。NAPI 将为大量的企业级多处理器平台带来帮助，它需要一个启用了 NAPI 的驱动程序。目前，很多驱动程序默认没有启用 NAPI，这就为调优网络子系统的性能提供了更广阔的空间。

6) 理解 Linux 调优参数

因为 Linux 是一个开源操作系统，所以有大量可用的性能监测工具。对这些工具的选择取决于个人喜好和对数据细节的要求。所有的性能监测工具都是按照同样的规则来工作的，所以无论使用哪种监测工具，都需要理解这些参数。下面列出了一些重要的参数，有效地理解它们大有裨益。

① 处理器参数

- **CPU utilization:** 这是一个很简单的参数，它直观地描述了每个 CPU 的利用率。在 xSeries 架构中，如果 CPU 的利用率长时间地超过 80%，就有可能出现处理器的瓶颈。
- **Runnable processes:** 描述了正在准备被执行的进程。在一个持续时间里，这个值不应该超过物理 CPU 数量的 10 倍，否则 CPU 方面就可能存在瓶颈。
- **Blocked:** 描述了那些因为等待 I/O 操作结束而不能被执行的进程，Blocked 可能指出用户正面临的 I/O 瓶颈。

- **User time**: 描述了处理用户进程的百分比, 包括 Nice time。如果 User time 的值很高, 则说明系统正在处理实际的工作。
- **System time**: 描述了 CPU 花费在处理内核操作 (包括 IRQ 和软件中断) 上的百分比。如果 System time 很高, 则说明系统可能存在网络或者驱动堆栈方面的瓶颈。一个系统通常只花费很少的时间去处理内核的操作。
- **Idle time**: 描述了 CPU 空闲的百分比。
- **Nice time**: 描述了 CPU 花费在处理 re-nicing 进程上的百分比。
- **Context switch**: 系统中线程之间进行交换的数量。
- **Waiting**: CPU 花费在等待 I/O 操作上的总时间。与 Blocked 相似, 一个系统不应该花费太多的时间在等待 I/O 操作上, 否则应该进一步检测 I/O 子系统是否存在瓶颈。
- **Interrupts**: Interrupts 值包括硬 Interrupts 和软 Interrupts, 硬 Interrupts 会对系统性能带来很多不利的影响。高 Interrupts 值指出系统可能存在一个软件的瓶颈, 可能是内核或者驱动程序。注意: Interrupts 值中包括 CPU 时钟导致的中断 (现代的 xServer 系统每秒会产生 1000 个 Interrupts 值)。

② 内存参数

- **Free memory**: 相比其他操作系统, Linux 空闲内存的值不应该作为一个性能参考的重要指标, 因为 Linux 内核会分配大量没有被使用的内存作为文件系统的缓存, 所以这个值通常比较小。
- **Swap usage**: 描述了已经被使用的 Swap 空间。
- **Buffer and cache**: 描述了为文件系统和块设备分配的缓存。在 Red Flag DC Server 5.0 版本中, 可以通过修改 /proc/sys/vm 中的 page_cache_tuning 来调整空闲内存中作为缓存的数量。
- **Slabs**: 描述了内核使用的内存空间, 注意内核的页面是不能被交换到磁盘上的。
- **Active versus inactive memory**: 提供了关于系统内存的 active 内存信息。inactive 内存是被 kswapd 守护进程交换到磁盘上的空间。

③ 网络参数

- **Packets received and sent**: 描述了一个指定网卡接收和发送的数据包的数量。
- **Bytes received and sent**: 描述了一个指定网卡接收和发送的数据包的字节数。
- **Collisions per second**: 提供了发生在指定网卡上的网络冲突的数量。持续出现这个值则表示在网络架构上出现了瓶颈, 而不是在服务器端出现了问题。在正常配置的网络中, 冲突是非常少见的, 除非用户的网络环境都是由 Hub 组成的。
- **Packets dropped**: 描述了被内核丢掉的数据包数量, 可能是因为防火墙或者网络缓存的缺乏。
- **Overruns**: 表达了超出网络接口缓存的次数。这个参数应该和 Packets dropped 一起来判断是否存在网络缓存或者网络队列过长方面的瓶颈。
- **Errors**: 记录了标志为失败的帧的数量。这可能是由错误的网络配置或者部分网线损坏导致的。在铜口千兆以太网环境中, 部分网线的损害是影响性能的一个重要因素。

④ 块设备参数

- Io wait: CPU 等待 I/O 操作所花费的时间。这个值持续很高通常是 I/O 瓶颈所导致的。
- Average queue length: I/O 请求的数量, 通常一个磁盘队列值为 2~3 为最佳情况, 更高的值则说明系统可能存在 I/O 瓶颈。
- Average wait: 响应一个 I/O 操作的平均时间。该值包括实际 I/O 操作的时间和在 I/O 队列里等待的时间。
- Transfers per second: 描述了每秒执行多少次 I/O 操作 (包括读和写)。Transfers per second 的值与 kBytes per second 的值结合起来可以用于估计系统的平均传输块大小, 当这个传输块大小和磁盘子系统的条带化大小相符合时, 可以获得最好的性能。
- Blocks read/write per second: 表达了每秒读/写的 Blocks 数量。在 2.6 内核中 Blocks 的大小是 1024Bytes, 在早些的内核版本中 Blocks 可以是不同的大小, 从 512Bytes 到 4KB。
- Kilo bytes per second read/write: 以 KB 为单位表示读/写块设备的实际数据的数量。

2. 常用 Linux 调优命令和工具

要实现对 Linux 的调优, 就需要用到一些 Linux 系统命令和工具来观察与监控系统的性能。下面介绍几个最常用的 Linux 调优命令和工具。

1) top 命令^①

top 命令经常用来监控 Linux 的系统状态, 如 CPU、内存的使用情况。下面通过一个运行中的 Web 服务器的 top 监控截图, 讲述 top 视图中各种数据的含义, 以及视图中各进程 (任务) 的字段的排序。

top 进入视图, 如图 6.17 所示。

```

文件(F) 编辑(E) 查看(V) 帮助(H)
top - 10:01:23 up 126 days, 14:29, 2 users, load average: 1.15, 1.42, 1.44
Tasks: 183 total, 1 running, 182 sleeping, 0 stopped, 0 zombie
Cpu(s): 6.7% us, 0.4% sy, 0.0% ni, 92.9% id, 0.0% wa, 0.0% hi, 0.0% si
Mem: 8306544k total, 7775876k used, 530668k free, 79236k buffers
Swap: 2031608k total, 2556k used, 2029052k free, 4231276k cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
14210 root        20   0 2446m 1.3g 34m  S   100  16.0 594:03.78 java
14183 root        20   0 2358m 1.2g 34m  S   12  15.3 592:43.04 java
22388 root        15   0 38848 18m 11m  S    0  0.2 60:39.38 gedit
10704 root        16   0 2652 1016 760  R    0  0.0 0:01.45 top
   1 root         0  0 3452 552 472  S    0  0.0 0:06.33 init
   2 root         RT  0 0 0 0 0  S    0  0.0 0:00.42 migration/0
   3 root        34  19 0 0 0  S    0  0.0 0:00.05 ksoftirqd/0
   4 root         RT  0 0 0 0 0  S    0  0.0 0:00.29 migration/1
   5 root        34  19 0 0 0  S    0  0.0 0:00.04 ksoftirqd/1
   6 root         RT  0 0 0 0 0  S    0  0.0 0:00.15 migration/2
   7 root        34  19 0 0 0  S    0  0.0 0:00.19 ksoftirqd/2
   8 root         RT  0 0 0 0 0  S    0  0.0 0:00.18 migration/3
   9 root        34  19 0 0 0  S    0  0.0 0:00.17 ksoftirqd/3
  10 root         RT  0 0 0 0 0  S    0  0.0 0:00.12 migration/4
  11 root        34  19 0 0 0  S    0  0.0 0:00.06 ksoftirqd/4
  12 root         RT  0 0 0 0 0  S    0  0.0 0:00.02 migration/5
  13 root        34  19 0 0 0  S    0  0.0 0:00.09 ksoftirqd/5
  14 root         RT  0 0 0 0 0  S    0  0.0 0:00.05 migration/6
  15 root        34  19 0 0 0  S    0  0.0 0:00.11 ksoftirqd/6
  16 root         RT  0 0 0 0 0  S    0  0.0 0:00.00 migration/7
  17 root        34  19 0 0 0  S    0  0.0 0:00.05 ksoftirqd/7

```

图 6.17

① 参考 <http://www.jb51.net/article/40807.htm>。

下面结合该视图讲解各个数据的含义。

第一行：10:01:23——当前系统时间；126 days,14:29——系统已经运行了 126 天 14 小时 29 分钟（在这期间没有重启过）；2 users——当前有两个用户登录系统；load average:1.15,1.42,1.44——load average 后面的 3 个数分别是 1 分钟、5 分钟、15 分钟的负载情况。load average 数据是每隔 5 秒钟检查一次活跃的进程数，然后按特定算法计算出的数值。如果这个数除以逻辑 CPU 的数量，结果高于 5，则表明系统正在超负荷运转。

第二行：Tasks——任务（进程），系统现在共有 183 个进程，其中处于运行状态的有 1 个，182 个正在休眠（sleep），处于 stopped 状态的有 0 个，处于 zombie（僵尸）状态的有 0 个。

第三行：CPU 状态。6.7% us——用户空间占用 CPU 的百分比；0.4% sy——内核空间占用 CPU 的百分比；0.0% ni——改变过优先级的进程占用 CPU 的百分比；92.9% id——空闲 CPU 百分比；0.0% wa——I/O 等待占用 CPU 的百分比；0.0% hi——硬中断（Hardware IRQ）占用 CPU 的百分比；0.0% si——软中断（Software Interrupts）占用 CPU 的百分比。

第四行：内存状态。8306544k total——物理内存总量（8GB）；7775876k used——使用中的内存总量（7.7GB）；530668k free——空闲内存总量（530MB）；79236k buffers——缓存的内存量（79MB）。

第五行：Swap 交换分区。2031608k total——交换区总量（2GB）；2556k used——使用的交换区总量（2.5MB）；2029052k free——空闲交换区总量（2GB）；4231276k cached——缓冲的交换区总量（4GB）。

这里需要说明的是，不能用 Windows 的内存概念理解这些数据，如果按 Windows 的方式，则此台服务器“危矣”：8GB 的内存总量只剩下 530MB 的可用内存。Linux 的内存管理有其特殊性，复杂点需要一本书来说明，这里只简单说一些和传统概念（Windows）的区别。第四行中使用中的内存总量（used）指的是现在系统内核控制的内存数，空闲内存总量（free）是内核还未纳入其管控范围的数量。纳入内核管理的内存并非都在使用中，还包括过去使用过的现在可以被重复利用的内存，内核并不把这些可被重新使用的内存交还到 free 中去，因此在 Linux 上 free 内存会越来越少，但不用为此担心。如果出于习惯去计算可用内存数，这里有一个近似的计算公式：第四行的 free+第四行的 buffers+第五行的 cached。依照这个公式，此台服务器的可用内存为：530668+79236+4231276=4.7GB。

对于内存监控，在 top 里我们要时刻监控第五行 Swap 交换分区的 used，如果这个数值在不断地变化，则说明内核在不断进行内存和 Swap 的数据交换，这表示真正的内存不够用了。

第六行是空行。

第七行以下：各进程（任务）的状态监控。PID——进程 ID；USER——进程所有者；PR——进程优先级；NI——nice 值，负值表示高优先级，正值表示低优先级；VIRT——进程使用的虚拟内存总量，单位为 KB（VIRT=SWAP+RESRES——进程使用的、未被换出的物理内存大小，单位为 KB）；RES=CODE+DATASHR——共享内存大小，单位为 KB；S——进程状态（D=不可中断的睡眠状态；R=运行；S=睡眠；T=跟踪/停止；Z=僵尸进程）；%CPU——上次更新到现在的 CPU 时间占用百分比；

%MEM——进程使用的物理内存百分比；TIME+——进程使用的 CPU 时间总计，单位为 1/100 秒；COMMAND——进程名称（命令名/命令行）。

多核 CPU 监控：在 top 基本视图中，按键盘数字“1”，即可监控每个逻辑 CPU 的状态，如图 6.18 所示。

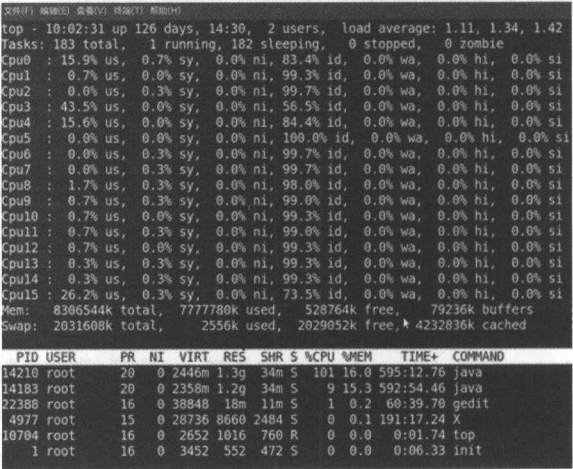


图 6.18

观察图 6.18，服务器有 16 个逻辑 CPU，实际上是 4 个物理 CPU。

进程字段排序：在进入 top 时，各进程默认是按照 CPU 的占用量来排序的，在【top 视图 01】中进程 ID 为 14210 的 Java 进程排在第一位（CPU 占用 100%），进程 ID 为 14183 的 Java 进程排在第二位（CPU 占用 12%）。可通过键盘指令来改变排序字段。比如，想监控哪个进程占用 MEM 最多，方法如下：

按“b”键（打开/关闭加亮效果），top 视图变化如图 6.19 所示。

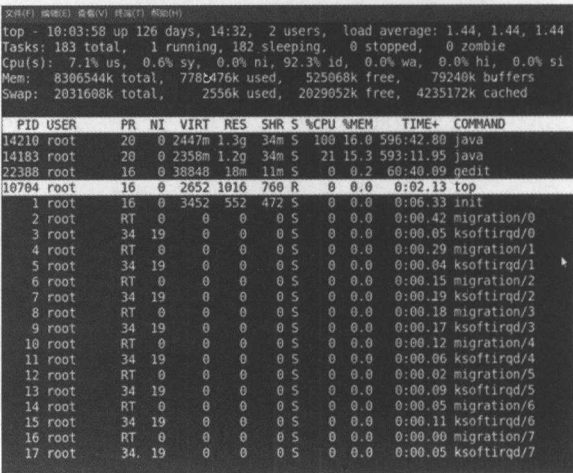


图 6.19

我们发现进程 ID 为 10704 的“top”进程被加亮了，top 进程就是视图第二行显示的唯一的运行态（running）的那个进程，可以通过按“y”键关闭或打开运行态进程的加亮效果。

按“x”键（打开/关闭排序列的加亮效果），top 视图变化如图 6.20 所示。

```

文本(F) 编辑(E) 查看(V) 终端(T) 帮助(H)
top - 10:05:02 up 126 days, 14:33, 2 users, load average: 1.43, 1.45, 1.45
Tasks: 183 total, 1 running, 182 sleeping, 0 stopped, 0 zombie
Cpu(s): 6.7% us, 0.5% sy, 0.0% ni, 92.7% id, 0.0% wa, 0.0% hi, 0.0% si
Mem: 8386544k total, 7783036k used, 523508k free, 79248k buffers
Swap: 2031608k total, 2556k used, 2029052k free, 4236204k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU  %MEM    TIME+  COMMAND
14210 root        20   0 2447m 1.3g 34m S 100.0 16.0 597:46.67 java
14183 root        20   0 2358m 1.2g 34m S 14.5 15.3 593:22.08 java
4977  root        15   0 28736 8660 2484 S 0.1 0.1 191:17.50 X
12848 root        16   0 393m 145m 31m S 0.1 1.8 15:21.39 java
22388 root        16   0 38848 18m 11m S 0.2 60:40.38 gedit
10704 root        16   0 2652 1016 760 R 0.0 0.0 0:02.41 top
1 root         0   0 3452 552 472 S 0.0 0.0 0:06.33 init
2 root         0   0 0 0 0 S 0.0 0.0 0:00:42 migration/0
3 root         0   0 0 0 0 S 0.0 0.0 0:00:05 ksoftirqd/0
4 root         0   0 0 0 0 S 0.0 0.0 0:00:29 migration/1
5 root         0   0 0 0 0 S 0.0 0.0 0:00:04 ksoftirqd/1
6 root         0   0 0 0 0 S 0.0 0.0 0:00:15 migration/2
7 root         0   0 0 0 0 S 0.0 0.0 0:00:19 ksoftirqd/2
8 root         0   0 0 0 0 S 0.0 0.0 0:00:18 migration/3
9 root         0   0 0 0 0 S 0.0 0.0 0:00:17 ksoftirqd/3
10 root        0   0 0 0 0 S 0.0 0.0 0:00:12 migration/4
11 root        0   0 0 0 0 S 0.0 0.0 0:00:06 ksoftirqd/4
12 root        0   0 0 0 0 S 0.0 0.0 0:00:02 migration/5
13 root        0   0 0 0 0 S 0.0 0.0 0:00:09 ksoftirqd/5
14 root        0   0 0 0 0 S 0.0 0.0 0:00:05 migration/6
15 root        0   0 0 0 0 S 0.0 0.0 0:00:11 ksoftirqd/6
  
```

图 6.20

可以看到，top 默认的排序列是“%CPU”。通过按“Shift+>”或“Shift+<”组合键可以向右或向左改变排序列。图 6.21 是按一次“Shift+>”组合键的效果图。

```

文本(F) 编辑(E) 查看(V) 终端(T) 帮助(H)
top - 10:05:54 up 126 days, 14:34, 2 users, load average: 1.89, 1.53, 1.47
Tasks: 183 total, 1 running, 182 sleeping, 0 stopped, 0 zombie
Cpu(s): 6.8% us, 0.5% sy, 0.0% ni, 92.7% id, 0.0% wa, 0.0% hi, 0.0% si
Mem: 8386544k total, 7784972k used, 521572k free, 79252k buffers
Swap: 2031608k total, 2556k used, 2029052k free, 4236720k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU  %MEM    TIME+  COMMAND
14210 root        20   0 2448m 1.3g 34m S 100.0 596:38.64 java
14183 root        20   0 2358m 1.2g 34m S 10.5 15.3 593:31.38 java
5711  root        16   0 492m 416m 34m S 0.1 5.1 157:54.62 java
12848 root        16   0 393m 145m 31m S 0.1 1.8 15:21.39 java
25104 root        16   0 53592 22m 11m S 0.3 0:07.23 nautilus
17996 root        16   0 58048 21m 16m S 0.3 0:00.65 kdeinit
22388 root        16   0 38848 18m 11m S 0.2 60:40.62 gedit
17916 root        16   0 50776 17m 9960 S 0.2 0:01.18 nautilus
25120 root        25  10 29532 16m 9860 S 0.2 1:08.63 rhn-applet-gui
17928 root        26  10 29592 16m 9860 S 0.2 1:09.48 rhn-applet-gui
17914 root        15   0 34256 15m 9152 S 0.2 0:01.70 gnome-panel
5483  gdm         16   0 60104 14m 7968 S 0.2 3:51.45 gdm-greeter
25102 root        15   0 33928 14m 8964 S 0.2 0:01.50 gnome-panel
17933 root        15   0 33580 11m 7936 S 0.1 0:00.39 mixer applet2
17899 root        15   0 16300 11m 2848 S 0.1 0:50.63 Xvnc
25118 root        15   0 33288 11m 7920 S 0.1 0:00.35 mixer applet2
25088 root        16   0 16012 11m 3172 S 0.1 20:54.46 Xvnc
25093 root        16   0 32388 10m 7112 S 0.1 0:00.25 gnome-session
17904 root        16   0 31704 10m 7116 S 0.1 0:00.25 gnome-session
17991 root        15   0 28152 9m 8360 S 0.1 0:00.90 kdeinit
17930 root        16   0 31572 9.9m 7528 S 0.1 0:00.19 wnck-applet
  
```

图 6.21

视图现在已经按照%MEM来排序了。

按“f”键，top 进入另一个视图，在这里可以编辑基本视图中的显示字段，如图 6.22 所示。

这里列出了所有可在 top 基本视图中显示的进程字段，有“*”并且标注为大写字母的字段是可显示的，没有“*”并且标注为小写字母的字段是不显示的。如果要在基本视图中显示“CODE”和“DATA”两个字段，可以分别按“r”和“s”键。

```

Current Fields: AEHI00TWNHBCDFGJPLRSUVYZX for window 1:Def
Toggle fields via field letter, type any other key to return

* A: PID      = Process Id
* E: USER     = User Name
* H: PR       = Priority
* I: NI       = Nice value
* O: VIRT     = Virtual Image (kb)
* Q: RES      = Resident size (kb)
* T: SHR      = Shared Mem size (kb)
* W: S        = Process Status
* K: %CPU     = CPU usage
* N: %MEM     = Memory usage (RES)
* M: TIME+    = CPU Time, hundredths
* B: PPID     = Parent Process Pid
* C: RUSER    = Real user name
* D: UID      = User Id
* F: GROUP    = Group Name
* G: TTY      = Controlling Tty
* J: AC       = Last used cpu (SMP)
* P: SWAP     = Swapped size (kb)
* L: TIME     = CPU Time
* R: CODE     = Code size (kb)
* S: DATA    = Data+Stack size (kb)
* U: nFLT     = Page Fault count
* V: nDRT     = Dirty Pages count
* Y: WCHAN    = Sleeping in Function
* Z: Flags    = Task Flags <sched.h>

* X: COMMAND  = Command name/line

Flags field:
0x00000001 PF_ALIGNWARN
0x00000002 PF_STARTING
0x00000004 PF_EXITING
0x00000040 PF_FORKNOEXEC
0x00000100 PF_SUPERPRIV
0x00000200 PF_DUMPCORE
0x00000400 PF_SIGNALED
0x00000800 PF_MEMALLOC
0x00002000 PF_FREE_PAGES (2.5)
0x00008000 debug flag (2.5)
0x00024000 special threads (2.5)
0x00100000 special states (2.5)
0x00100000 PF_USEDPU (thru 2.4)

```

图 6.22

按回车键返回基本视图，可以看到多了“CODE”和“DATA”两个字段，如图 6.23 所示。

```

top - 10:13:29 up 126 days, 14:41, 2 users, load average: 2.52, 1.85, 1.63
Tasks: 183 total, 1 running, 182 sleeping, 0 stopped, 0 zombie
Cpu(s): 6.9% us, 0.4% sy, 0.0% ni, 92.7% id, 0.0% wa, 0.0% hi, 0.0% si
Mem: 8306544k total, 7796948k used, 509596k free, 79128k buffers
Swap: 2031608k total, 2556k used, 2029052k free, 4245164k cached

  PID USER      PR  NI  VIRT  RES  SHR  S %CPU  %MEM    TIME+  CODE DATA COMMAND
14210 root        20   0 2451m 1.2g 34m S   100 16.1 686:15.93 48 2230 java
14183 root        20   0 2359m 1.2g 34m S   15.3 594:52.61 48 2.30 java
5711 root        16   0 492m 416m 34m S    0 5.1 157:54.74 48 446m java
12848 root       16   0 393m 145m 31m S    0 1.8 15:21.86 48 351m java
25104 root       16   0 53592 22m 11m S    0 0.3 0:07.23 588 22m nautilus
17996 root       16   0 58048 21m 16m S    0 0.3 0:00.65 36 16m kdeinit
22388 root       15   0 38848 18m 11m S    0 0.2 60:42.72 408 7780 gedit
17916 root       16   0 50776 17m 9960 S    0 0.2 0:01.18 588 20m nautilus
25120 root       25 10 29532 16m 9860 S    0 0.2 1:08.67 0 5976 rhn-applet-gui
17928 root       25 10 29592 16m 9860 S    0 0.2 1:09.51 0 6036 rhn-applet-gui
17914 root       15   0 34256 15m 9152 S    0 0.2 0:01.70 436 6328 gnome-panel
5483 gdm         15   0 60104 14m 7968 S    0 0.2 3:51.45 136 7136 gdm-greeter
25102 root       15   0 33928 14m 8964 S    0 0.2 0:01.50 436 6200 gnome-panel
17933 root       15   0 33580 11m 7936 S    0 0.1 0:00.39 28 5160 mixer_applet2
17899 root       15   0 16300 11m 2848 S    0 0.1 0:50.65 1568 10m Xvnc
25118 root       15   0 33288 11m 7920 S    0 0.1 0:00.35 28 4868 mixer_applet2
25088 root       15   0 16012 11m 3172 S    0 0.1 20:55.14 1568 9608 Xvnc
25093 root       16   0 32388 10m 7112 S    0 0.1 0:00.25 120 5332 gnome-session
17898 root       16   0 31704 10m 7116 S    0 0.1 0:00.25 120 4648 gnome-session

```

图 6.23

top 命令是 Linux 上进行系统监控的首选命令，但有时却达不到我们的要求，如当前这台服务器，top 监控就有很大的局限性。top 命令的监控最小单位是进程，所以看不到我们关心的 Java 线程数和客户连接数，而这两个指标是 Java 的 Web 服务非常重要的指标，通常可以用 ps 和 netstat 两个命令来补充 top 命令的不足。

监控 Java 线程数：

```
ps -eLf|grep java|wc -l
```

监控网络客户连接数：

```
netstat -n|grep tcp|grep 侦听端口|wc -l
```

上述两个命令可改变 grep 的参数来达到更细致的监控要求。在 Linux 系统“一切都是文件”的思想贯彻指导下，所有进程的运行状态都可以用文件来获取。在系统根目录/proc 中，每个数字子目录的名字都是运行中的进程的 PID，进入任一个进程目录，都可以通过其中的文件或目录来观察进程的各项运行指标，例如，task 目录就是用来描述进程中的线程的，因此，也可以通过下面的方法来获

取某进程中运行的线程数量（PID 指的是进程 ID）：

```
ls /proc/PID/task|wc -l
```

在 Linux 中还有一个命令 `pmap`，可以用来输出进程内存的状态，也可以用来分析线程堆栈，如下：

```
pmap PID
```

2) vmstat 命令^①

`vmstat` 命令用来获取有关进程、虚拟内存、页面交换空间及 CPU 活动的信息，这些信息反映了系统的负载情况。

在系统中运行的每个进程都需要使用内存，但不是每个进程都需要每时每刻使用系统分配的内存空间。当系统运行所需内存超过实际的物理内存时，内核会释放某些进程所占但并未使用的部分或所有物理内存，将这部分资料存储在磁盘上直到进程下一次被调用，并将释放出来的内存提供给有需要的进程使用。

在 Linux 内存管理中，主要通过“调页（Paging）”和“交换（Swapping）”来完成上述的内存调度。调页算法是将内存中最近不常使用的页面交换到磁盘上，把活动页面保留在内存中供进程使用。交换技术是将整个进程，而不是部分页面，全部交换到磁盘上。

分页（Page）写入磁盘的过程被称作 Page-Out，分页（Page）从磁盘重新回到内存的过程被称作 Page-In。当内核需要一个分页时，发现此分页不在物理内存中（因为已经被 Page-Out 了），此时就发生了分页错误（Page Fault）。

当系统内核发现可运行内存减少时，就会通过 Page-Out 来释放一部分物理内存。尽管 Page-Out 不经常发生，但是如果 Page-Out 频繁地发生，直到当内核管理分页的时间超过运行程式的时间时，系统效能就会急剧下降。这时的系统运行已经非常缓慢或进入暂停状态，这种状态也被称作 Thrashing（颠簸）。

① vmstat 命令的使用方法

```
vmstat [-a] [-n] [-S unit] [delay [ count]]
vmstat [-s] [-n] [-S unit]
vmstat [-m] [-n] [delay [ count]]
vmstat [-d] [-n] [delay [ count]]
vmstat [-p disk partition] [-n] [delay [ count]]
vmstat [-f]
vmstat [-V]
```

各参数的含义如下。

- `-a`：显示活跃和非活跃内存。
- `-f`：显示从系统启动至今的 fork 进程数量。
- `-m`：显示 `/proc/slabinfo`。
- `-n`：只在开始时显示一次各字段名称。

^① 参考 <http://blog.chinaunix.net/uid-22741583-id-3087675.html>。

- -s: 显示内存相关统计信息及多种系统活动数量。
- delay: 刷新时间间隔。如果不指定, 则只显示一条结果。
- count: 刷新次数。如果不指定刷新次数, 但指定了刷新时间间隔, 这时刷新次数为无穷次。
- -d: 显示磁盘相关统计信息。
- -p: 显示指定磁盘分区统计信息。
- -S: 使用指定单位显示。参数有 k、K、m、M, 分别代表 1000、1024、1 000 000、1 048 576 字节 (Byte)。默认单位为 K (1024Bytes)。
- -V: 显示 vmstat 版本信息。

② 字段含义说明

字段含义说明如表 6.1 所示。

表 6.1

类 别	项 目	含 义	说 明
Procs (进程)	r	等待执行的任务数	显示了正在执行和等待 CPU 资源的任务个数。当这个值超过 CPU 个数时, 就会出现 CPU 瓶颈
	b	置于等待队列 (等待资源, 等待输入/输出) 的内核线程数目	
Memory (内存)	swpd	正在使用的虚拟内存大小, 单位为 k	
	free	空闲内存大小	
	buff	已用的 buff 大小, 对块设备的读/写进行缓冲	
	cache	已用的 cache 大小, 文件系统的 cache	
	inact	非活跃内存大小, 即被标明可回收的内存, 区别于 free 和 active	具体含义见概念补充 (当使用-a 选项时显示)
	active	活跃的内存大小	具体含义见概念补充 (当使用-a 选项时显示)
	pi	从调页空间调度进的页面	
	po	调出到调页空间的页面	
Swap	si	每秒从交换区写入内存的大小 (单位为 kbit/s)	
	so	每秒从内存写到交换区的大小	
I/O	bi	每秒读取的块数 (读磁盘)	现在的 Linux 版本, 块的大小为 1024Bytes
	bo	每秒写入的块数 (写磁盘)	
system	in	每秒中断数, 包括时钟中断	这两个值越大, 看到的由内核消耗的 CPU 时间会越多
	cs	每秒上下文切换数	
CPU (以百分比表示)	us	用户进程执行消耗 CPU 时间(usertime)	us 的值比较高时, 说明用户进程消耗的 CPU 时间多。但是如果长期超过 50%地使用, 那么就应该考虑优化程序算法或其他措施了

续表

类 别	项 目	含 义	说 明
CPU (以百分比表示)	sy	系统进程消耗 CPU 时间(systemtime)	sy 的值过高时,说明系统内核消耗的 CPU 资源多。这个不是良性的表现,我们应该检查原因
	Id	空闲时间(包括 I/O 等待时间)	
	wa	等待 I/O 时间	wa 过高时,说明 I/O 等待比较严重。这可能是由于磁盘大量随机访问造成的,也有可能是磁盘的带宽出现瓶颈

③ 常见问题处理

• CPU 瓶颈判断和解决办法。

- (1) 如果 r 经常大于 4, 且 id 经常小于 40, 则表示 CPU 的负荷很严重。
- (2) 如果 pi、po 长期不等于 0, 则表示内存不足。
- (3) 如果 disk 经常不等于 0, 且在 b 中的队列大于 3, 则表示 I/O 性能不佳。
- (4) 如果在 processes 中运行的序列(processsr)是连续的, 且大于在系统中的 CPU 个数, 则表示系统现在运行比较慢, 有很多进程正在等待 CPU。

(5) 如果 r 的输出数大于系统中可用 CPU 个数的 4 倍, 则系统面临 CPU 资源短缺的问题, 或者 CPU 的速率过低, 系统中有很多进程正在等待 CPU, 造成系统中进程运行过慢。

(6) 如果空闲时间(cpu id)持续为 0, 并且系统时间(cpu sy)是用户时间的两倍(cpu us), 则系统面临 CPU 资源短缺的问题。

解决办法:

当发生以上问题时, 请先调整应用程序对 CPU 的占用情况, 使应用程序能够更有效地使用 CPU, 同时可以考虑增加更多的 CPU。关于 CPU 的使用情况, 还可以结合 mpstat、ps aux、top、prstat -a 等相应的命令来综合考虑关于具体 CPU 的使用情况, 以及哪些进程正在大量占用 CPU 时间。一般情况下, 应用程序的问题会比较大, 如一些 SQL 语句不合理等都会造成这种现象。

• 内存瓶颈判断和解决办法。

内存的瓶颈是由 scanrate(sr)来决定的。scanrate 是通过每秒的时钟算法来进行页扫描的。如果 scanrate(sr)连续大于每秒 200 页, 则表示可能存在内存缺陷。同样, page 项中的 pi 和 po 两栏表示页面每秒调入的页数和每秒调出的页数, 如果该值经常为非零值, 则也有可能存在内存瓶颈。当然, 如果个别的时候不为 0, 则属于正常的页面调度。这就是虚拟内存的主要原理。

解决办法:

- (1) 调节 applications&servers, 使得对内存和 cache 的使用更加有效。
- (2) 增加系统的内存。

关于内存的使用情况, 还可以结合 ps aux、top、prstat -a 等相应的命令来综合考虑关于具体内存的使用情况, 以及哪些进程正在占用大量的内存。

一般情况下, 如果系统显示内存的占用率比较高, 但是同时 CPU 的占用率却很低, 则可以考虑

有很多应用程序占用了内存而没有释放。这时应该考虑让未占用 CPU 时间的应用程序或一些后台的程序释放占用的内存。

3) iostat 命令^①

iostat 命令可以给我们提供丰富的 I/O 状态数据。

① 基本使用

```
$iostat -d -k 1 10
```

参数-d 表示设备（磁盘）的使用状态；-k 表示某些以 Block 为单位的列强制使用 Kilobytes 为单位；1 10 表示数据显示每隔 1 秒刷新一次，共显示 10 次。示例代码如下：

```
$iostat -d -k 1 10
Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda                 39.29      21.14        1.44  441339807  29990031
sda1                 0.00        0.00        0.00    1623      523
sda2                 1.32        1.43        4.54  29834273  94827104
sda3                 6.30        0.85       24.95  17816289  520725244
sda5                 0.85        0.46        3.40   9543503  70970116
sda6                 0.00        0.00        0.00    550      236
sda7                 0.00        0.00        0.00    406        0
sda8                 0.00        0.00        0.00    406        0
sda9                 0.00        0.00        0.00    406        0
sda10                60.68      18.35       71.43 383002263 1490928140
```

```
Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda                 327.55     5159.18     102.04    5056      100
sda1                 0.00        0.00        0.00        0        0
```

tps: 该设备每秒的传输次数 (Indicate the number of transfers per second that were issued to the device)。“一次传输”意思是“一次 I/O 请求”。多个逻辑请求可能会被合并为“一次 I/O 请求”。“一次传输”请求的大小是未知的。

kB_read/s: 每秒从设备读取的数据量；kB_wrtn/s: 每秒向设备写入的数据量；kB_read: 读取的总数据量；kB_wrtn: 写入的总数据量；这些值的单位都为 Kilobytes。

在上面的例子中，可以看到磁盘 sda 及它的各个分区的统计数据，当时统计的磁盘总 TPS 是 39.29，下面是各个分区的 TPS（因为是瞬间值，所以总 TPS 并不严格等于各个分区 TPS 的总和）。

② -x 参数

使用-x 参数可以获得更多的统计信息。

```
iostat -d -x -k 1 10
Device:  rrqm/s wrqm/s  r/s  w/s  rsec/s  wsec/s   rkB/s   wkB/s  avgrq-sz
avgqu-sz  await  svctm  %util
sda       1.56 28.31  7.80 31.49  42.51   2.92   21.26   1.46   1.16
0.03  0.79  2.62 10.28
Device:  rrqm/s wrqm/s  r/s  w/s  rsec/s  wsec/s   rkB/s   wkB/s  avgrq-sz
```

^① 参考 <http://www.orczhou.com/index.php/2010/03/iostat-detail/>。

```
avgqu-sz  await  svctm  %util
sda      2.00  20.00  381.00  7.00  12320.00  216.00  6160.00  108.00  32.31
1.75     4.50   2.17  84.20
```

rrqm/s: 每秒这台设备相关的读取请求有多少被合并了(当系统调用需要读取数据的时候, VFS 将请求发送到各个 FS, 如果 FS 发现不同的读取请求读取的是相同 Block 的数据, 那么 FS 会将这些请求合并)。

wrqm/s: 每秒这台设备相关的写入请求有多少被合并了。

rsec/s: 每秒读取的扇区数。

wsec/s: 每秒写入的扇区数。

r/s: 每秒读请求。

w/s: 每秒写请求。

await: 每个 I/O 请求处理的平均时间(单位是微秒或毫秒)。这里可以理解为 I/O 的响应时间。一般情况下, 系统 I/O 响应时间应该低于 5ms。

%util: 在统计时间内所有处理 I/O 时间除以总统计时间。例如, 如果统计间隔为 1s, 该设备有 0.8s 在处理 I/O, 而有 0.2s 在闲置, 那么该设备的 %util=0.8/1=80%, 所以该参数暗示了设备的繁忙程度。一般地, 如果该参数是 100%, 则表示设备已经接近满负荷运行了(当然, 如果是多磁盘, 即使 %util 为 100%, 由于磁盘的并发能力, 磁盘使用未必达到瓶颈)。

③ -c 参数

iostat 命令还可以用来获取 CPU 部分状态值, 如下:

```
iostat -c 1 10
avg-cpu:  %user   %nice   %sys   %iowait  %idle
          1.98    0.00    0.35   11.45   86.22
avg-cpu:  %user   %nice   %sys   %iowait  %idle
          1.62    0.00    0.25   34.46   63.67
```

④ 常见用法

```
$iostat -d -k 1 10          #查看 TPS 和吞吐量信息
iostat -d -x -k 1 10        #查看设备使用率(%util)、响应时间(await)
iostat -c 1 10              #查看 cpu 状态
```

⑤ 实例分析

```
$$iostat -d -k 1 |grep sda10
Device:      tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda10        60.72      18.95        71.53  395637647  1493241908
sda10        299.02     4266.67      129.41    4352      132
sda10        483.84     4589.90     4117.17    4544     4076
sda10        218.00     3360.00     100.00    3360      100
sda10        546.00     8784.00     124.00    8784      124
sda10        827.00    13232.00     136.00   13232     136
```

可以看到, 磁盘每秒传输次数平均约为 400; 每秒磁盘读取约 5MB, 写入约 1MB。

```
iostat -d -x -k 1
```

```

Device:   rrqm/s wrqm/s   r/s    w/s   rsec/s wsec/s   rkB/s   kB/s avgrq-sz
avgqu-sz await  svctm  %util
sda      1.56 28.31  7.84 31.50  43.65   3.16   21.82   1.58   1.19
0.03    0.80  2.61 10.29
sda      1.98 24.75 419.80  6.93 13465.35 253.47 6732.67 126.73 32.15
2.00    4.70  2.00 85.25
sda      3.06 41.84 444.90 54.08 14204.08 2048.98 7102.04 1024.49 32.57
2.10    4.21  1.85 92.24

```

可以看到，磁盘的平均响应时间小于 5ms，磁盘使用率大于 80%。这说明磁盘响应正常，但是已经很繁忙了。

4) nmon 工具使用^①

前面介绍的都是零散的命令，nmon 是分析 AIX 和 Linux 性能的免费工具。它综合收集系统的信息，以图形化的形式展现出来，方便系统管理对性能的分析。

(1) 下载 nmon。根据 CPU 的类型选择下载相应的版本。

(2) 初始化 nmon 工具。根据不同的平台，初始化对应平台的 nmon 工具，然后直接运行 nmon 即可。

直接运行 nmon 可以实时监控系统资源的使用情况，执行下面的步骤可以展现一段时间内系统资源消耗的报告。

如图 6.24 所示是直接执行 nmon 命令实时监控系统资源消耗情况的截图。可以看到，CPU、内存、磁盘和网络的消耗情况都被很直观地展现出来。

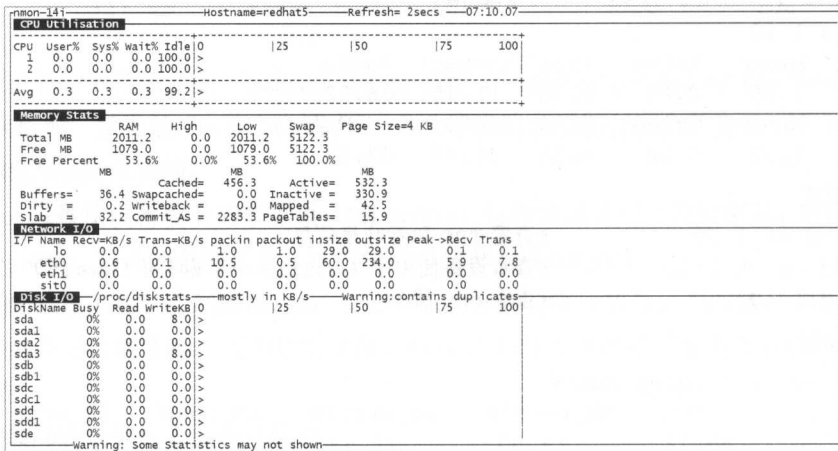


图 6.24

(3) 生成 nmon 报告。

a. 采集数据：

```
#nmon - s10 -c60 -f -m /home/
```

^① Nmon 性能——分析 AIX 和 Linux 性能的免费工具：http://www.ibm.com/developerworks/cn/aix/library/analyze_aix/index.html。

Nmon Analyser——生成 AIX 性能报告的免费工具：http://www.ibm.com/developerworks/cn/aix/library/nmon_analyser/index.html。

参数解释:

- -s10: 每 10 秒采集一次数据。
- -c60: 采集 60 次, 即采集 10 分钟的数据。
- -f: 生成的数据文件名中包含文件创建的时间。
- -m: 生成的数据文件的存放目录。

这样就会生成一个 nmon 文件, 每 10 秒更新一次, 直到 10 分钟后。生成的文件名如 <hostname>_090824_1306.nmon, 其中 “hostname” 是这台主机的名称。

使用 nmon -h 命令可以查看更多帮助信息。

b. 生成报表, 下载 nmon analyser (生成性能报告的免费工具)。

将之前生成的 nmon 数据文件上传到 Windows 机器中, 用 Excel 打开分析工具 nmonanalyserv33C.xls。单击 Excel 文件中的 “Analyzenmondata” 按钮, 选择 nmon 数据文件, 这样就会生成一个分析后的结果文件 hostname_090824_1306.nmon.xls, 用 Excel 打开生成的文件就可以看到结果。

如果宏不能运行, 则需要执行以下操作: 工具→宏→安全性→中, 然后打开文件并允许运行宏。

如图 6.25 所示是在测试环境中生成的 nmon 报告截图, 图中画框区域为不同指标的分析报告。

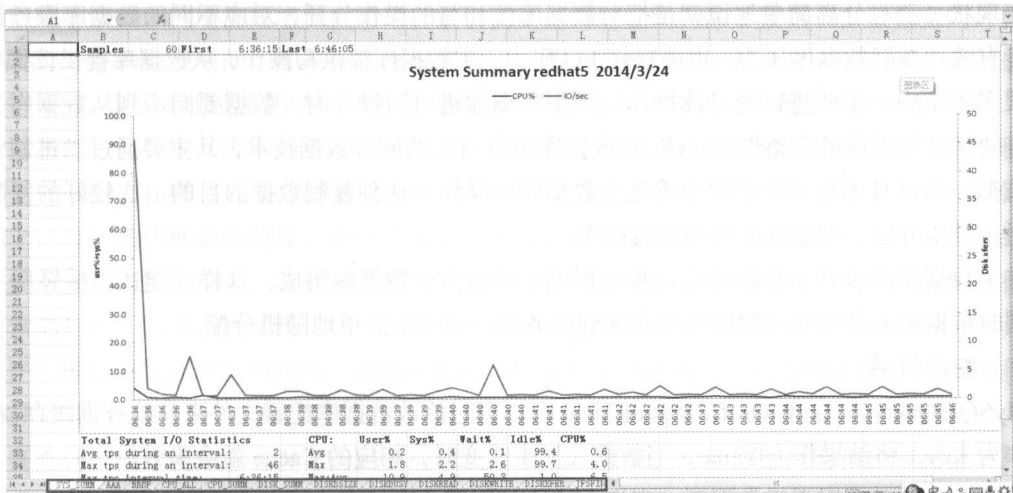


图 6.25

c. 自动按天采集数据。

在 crontab 中增加一条记录:

```
0 0 * * * root nmon -s300 -c288 -f -m /home/ > /dev/null 2>&1
```

300×288=86 400 (秒), 正好是一天的数据。

3. 数据库调优思路^①

数据库调优和具体的数据库强相关, 需要综合的计算机知识。这里介绍一些通用的思路, 要想

^① 参见 <http://jameswxx.iteye.com/blog/591504>。

真正成为数据库调优高手，还需要结合具体的数据库知识和长期的实战经验。

调优是门技术活，需要建立一个科学的方法。在动手调优前，首先要记住调优的原则，动手测试基准值，并且做好备份，每次调优只改动一个参数，只有这样才知道每个参数的调优效果。

调优一般分业务调优、数据库调优、操作系统调优等几大块，调优一定是一个端到端的、系统的优化过程。越是上层业务，一般来说越有效果。所以对于一个稳定的业务系统来说，千万不要第一时间去调整数据库或者系统参数，而是要彻底理解业务逻辑。一般来说，自顶向下调优效果最明显。

下面介绍一些常见的调优思路。

1) 业务优化

数据库调优，业务上的优化绝对是第一位的，越是底层的東西越稳定，所有出现了数据库运行缓慢的场景，不要一上来就期望通过调整参数来达到效果，性能低下往往是使用不当导致的。

2) 读写分离

这在 Web 应用中非常普遍。随着一个网站的业务不断扩展，数据量不断增加，数据库的压力也会越来越大，对数据库或者 SQL 的基本优化可能达不到最终的效果，这时可以采用读写分离的策略来改变现状。读写分离简单地讲就是把对数据库读和写的操作分开，对应不同的数据库服务器，这样既能有效地减轻数据库压力，也能减轻 I/O 压力。主数据库提供写操作，从数据库提供读操作，其实在很多系统中，主要进行的是读操作。当主数据库进行写操作时，数据要同步到从数据库，这样才能有效保证数据库的完整性。MySQL 数据库也有自己的同步数据技术，其主要通过二进制日志来复制数据，通过日志在从数据库中重复主数据库的操作来达到复制数据的目的。比较好的复制策略就是通过异步方法，把数据同步到从数据库。

在主数据库同步到从数据库后，从数据库一般由多台数据库组成，这样才能达到减轻压力的目的。同时根据服务器的压力把读操作分配到服务器，而不是简单地随机分配。

3) 查询缓存

MySQL QueryCache 和 Oracle ResultCache 可以通过修改数据库配置文件来实现查询缓存，将 SQL 语句作为 Key，将结果作为 Value，当数据表发生改变时，相应的 Cache 就会失效。

适用场合：对数据不经常更新，查询方式比较固定（支持表连接，但不支持函数）。

4) 分表分区

对于一些超大型的表，分区是非常有用的。分区是一种逻辑概念，和 Oracle 的分区概念是一样的。在通常情况下，一张表就是一个整体，当发生数据访问的时候，也就是对整张表或整张表的索引进行访问。所谓分区，通俗点讲，就是把表按一定的规律划分成更小的逻辑单位，当发生访问的时候，不以表为单位进行访问，而是先在表的基础上判断数据在哪个分区，然后对特定的分区进行访问。正确的分区有利于提高查询性能。

5) 中间表

将原数据根据想得到的目标数据进行一系列的处理，得出一套中间表，直接从中间表中进行查

询,通过定时调度定时更新中间表。

适用场合:对数据内容实时性要求不高,如数据分析。

6) 索引

索引是各种关系型数据库系统中最常见的一种逻辑单元,是关系型数据库系统举足轻重的组成部分,对于提高检索数据速度有着至关重要的作用。索引的原理是根据索引值得到行指针,然后快速定位到数据库记录。索引的使用是一把双刃剑,一定要适当。

为什么要创建索引?这是因为,创建索引可以大大提高系统的性能:第一,通过创建唯一性索引,可以保证数据库表中每一行数据的唯一性;第二,可以大大加快数据的检索速度,这也是创建索引的最主要原因;第三,可以加速表和表之间的连接,特别是在实现数据的参考完整性方面特别有意义;第四,在使用分组和排序子句进行数据检索时,同样可以显著减少查询中分组和排序的时间;第五,通过使用索引,可以在查询过程中使用优化隐藏器,从而提高系统的性能。

也许有人会问:增加索引有如此多的优点,为什么不对表中的每一列创建一个索引呢?这种想法固然有其合理性,但也有其片面性。虽然索引有许多优点,但是为表中的每一列都增加索引,是非常不明智的。这是因为,增加索引也有许多不利的方面:第一,创建索引和维护索引要耗费时间,这种时间随着数据量的增加而增加;第二,索引需要占物理空间,除了数据表占数据空间外,每个索引还要占一定的物理空间,如果要建立聚簇索引,那么需要的空间就会更大;第三,当对表中的数据进行增加、删除和修改操作时,索引也要动态维护,这样就降低了数据的维护速度。

索引是建立在数据库表中的某些列上面的。因此,在创建索引的时候,应该仔细考虑在哪些列上可以创建索引,在哪些列上不能创建索引。一般来说,应该在哪些列上创建索引:在经常需要搜索的列上,可以加快搜索的速度;在作为主键的列上,强制该列的唯一性和组织表中数据的排列结构;在经常用在连接的列上,这些列主要是一些外键,可以加快连接的速度;在经常需要根据范围进行搜索的列上,因为索引已经排序,其指定的范围是连续的;在经常需要排序的列上,因为索引已经排序,可以加快排序查询时间;在经常使用 WHERE 子句的列上,可以加快条件的判断速度。

同样,对于有些列不应该创建索引。一般来说,不应该创建索引的列具有下列特点:第一,对于那些在查询中很少使用或者参考的列不应该创建索引;第二,对于那些只有很少数据值的列也不应该创建索引;第三,对于那些定义为 text、image 和 bit 数据类型的列不应该创建索引;第四,当修改性能远远大于检索性能时,不应该创建索引。

7) SQL 优化

一条 SQL 语句大约要经过三个阶段:编译优化、执行和取值。很多人不知道 SQL 语句在 SQL Server 中是如何执行的,他们担心自己所写的 SQL 语句会被 SQL Server 误解。

事实上,这样的担心是不必要的。在 SQL Server 中有一个“查询分析优化器”,它可以计算出 WHERE 子句中的搜索条件,并确定哪个索引能缩小表扫描的搜索空间。也就是说,它能实现自动优化。

8) 系统调优

系统调优就是通过调整硬件（RAID 卡、网卡等）参数及操作系统等各种系统参数来实现数据库最大性能。比如 RAID 卡常用参数：readahead, always writeback with bbu, diskcache enable, stripesize 128K。各参数的含义如下。

- readahead: 预读扇区的内容，适合请求时连续的，所有请求都是随机的情况下选择 No-read-ahead 更合适。
- always writeback with bbu: RAID 卡配置有电池时使用的选项。在写磁盘的时候，先写入 RAID 卡的 cache，然后由 cache 写入磁盘中，因而会提高写盘速度。
- stripesize 128K: 条带参数设置，一般情况下数据库推荐值和表的一条记录大小一致。

4. Greenplum 实战安装

1) 操作系统准备（所有机器都需要如此安装）

① 磁盘分区（RAID）

在 RH2285+1078 卡配置情况下，建议用两块磁盘做 raid1，用于操作系统的安装，剩余的磁盘做成一个 raid10。调整 RAID 卡参数：read ahead, always write back with bbu, disk cache enable, stripe size 128K。

② 操作系统安装

- 系统目录全部挂载到第一个盘 sda 上。分配原则为：Swap 区，内存的 2 倍；/boot 区，512MB。
- /sda 挂载所有第一块盘的剩余空间。
- /sdb 删除，不格式化，操作系统安装成功后手动安装 XFS 文件系统。

③ 安装 XFS

安装过程如下：

```
mkdir /data
```

```
example about how to create xfs:
```

```
-----
# rpm -ivh ./xfsprogs-2.9.4-4.el5.x86_64.rpm
# /sbin/parted -s /dev/sdb mklabel gpt
# /sbin/parted -s /dev/sdb mkpart primary 0 100% 分区，GPT 分区比 MBR 支持更大的空间
# /sbin/mkfs.xfs -f -L /data /dev/sdb1 创建 XFS 文件系统，-L 表示设置 label
```

```
example about mkfs.xfs:
```

```
-----
mkfs.xfs -f -i size=512,attr=2 -l size=128m,lazy-count=1 -d su=64k,sw=5 -L /data /dev/sdb1
```

```
-l size=128m, 指定元数据日志大小
```

```
example about xfs with /etc/fstab:
```

```
-----
```



```

LABEL=/data /data xfs rw,noatime,inode64,allocsize=16m 0 0
rw:读写平衡模式
noatime,nodiratime: 关闭 atime 更新
inode64,inode 记录文件系统数据结构, 64 表示 64 位, 支持更多的文件
allocsize 缓冲 I/O 文件结束时预分配大小

```

```
mount -a
```

```
Add in, /etc/rc.local
```

```
-----
/sbin/blockdev --setra 16384 /dev/sd*    磁盘一次预读的大小 (KB)
```

④ 服务配置 (所有机器有效)

(1) 关闭防火墙。

a. RedHat 安装时, 不要安装防火墙。

b. 以 root 用户登录, 检查 iptables 是否关闭。

```
chkconfig --list|grep iptables
```

如果打开 iptables, 则使用如下命令关闭。

```

chkconfig --level 2 iptables off
chkconfig --level 3 iptables off
chkconfig --level 4 iptables off
chkconfig --level 5 iptables off

```

(2) 修改 inittab。

修改/etc/inittab 文件。

```
vi /etc/inittab
```

其中有一行 “id:5:initdefault:”, 将其中的 5 修改为 3。

Linux 启动分不同的级别。5 表示进入 GUI 模式, 3 表示进入多用户命令行模式。

(3) 修改/etc/sysctl.conf。

```
vi /etc/sysctl.conf
```

最终结果如下 (2285 机器建议直接覆盖即可):

```

# Kernel sysctl configuration file for Red Hat Linux
kernel.sem = 250 512000 100 2048
kernel.shmmax = 500000000
kernel.shmmni = 4096
kernel.shmall = 4000000000
kernel.sysrq = 1
kernel.core_uses_pid = 1
kernel.msgmni = 2048
kernel.msgmnb = 65536
kernel.msgmax = 65536
net.core.rmem_default = 262144
net.core.wmem_default = 262144
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216

```

```

net.core.netdev_max_backlog = 10000
net.ipv4.tcp_rmem = 4096 65536 16777216
net.ipv4.tcp_wmem = 4096 65536 16777216
net.ipv4.tcp_timestamps = 0
net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_tw_recycle=1
net.ipv4.tcp_max_syn_backlog=4096
net.ipv4.conf.default.accept_source_route = 0
net.ipv4.conf.all.arp_filter = 1
net.ipv4.conf.default.arp_filter = 1
net.ipv4.ip_forward = 0
net.ipv4.ip_local_port_range = 1025 65535
vm.overcommit_memory=2
vm.min_free_kbytes = 65536
fs.aio-max-nr = 1048576
fs.file-max = 6815744

```

修改完成后，以 root 用户运行 `sysctl -p`，检查是否有错误，并使参数生效。

`sysctl` 配置的是系统参数，上面调整的是网络、内核、内存等参数。

(4) 修改/etc/security/limits.conf。

在文件末尾添加如下内容：

```

* soft nofile 65536
* hard nofile 65536
* soft nproc 131072
* hard nproc 131072

```

解除 Linux 系统的最大进程数和最大文件数的打开限制。其中，* 代表所有用户，`nofile` 代表打开文件的数量，`nproc` 代表打开进程的数量。

(5) 修改/etc/pam.d/login。

在文件末尾添加如下内容：

```

session required /lib64/security/pam_limits.so
session required pam_limits.so

```

这是 Linux 操作系统的登录配置文件。

(6) 修改 grub。

编辑 `/boot/grub/menu.lst`（如果存在 `/boot/grub/grub.conf` 则一并修改），在第一个 Kernel 后面加上：
`elevator=deadline`

执行 `boot/grub # cat menu.lst`，出现如下结果，添加其中的黑色粗体部分：

```

# Modified by YaST2. Last modification on Tue Jul  5 12:30:23 CST 2011
default 0
timeout 8
##YaST - generic_mbr
gfxmenu (hd0,0)/message
##YaST - activate

```

```

###Don't change this comment - YaST2 identifier: Original name: linux###

```

```

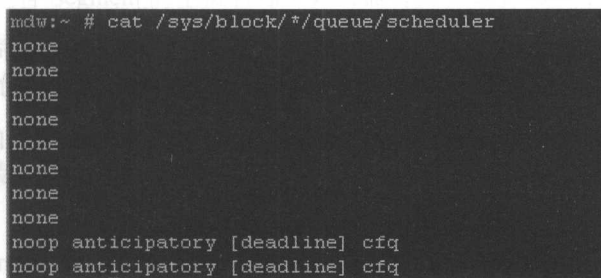
title SUSE Linux Enterprise Server 11 SP1 - 2.6.32.12-0.7
    root (hd0,0)
    kernel /vmlinuz-2.6.32.12-0.7-default
root=/dev/disk/by-id/scsi-3680fb_06352c1e00015a4dd99a08bbc6e-part2
resume=/dev/disk/by-id/scsi-3680fb06352c1e_00015a4dd99a08bbc6e-part3 splash=silent
crashkernel=256M-:128M showopts vga=0x317Q elevator=deadline
    initrd /initrd-2.6.32.12-0.7-default

###Don't change this comment - YaST2 identifier: Original name: failsafe###
title Failsafe -- SUSE Linux Enterprise Server 11 SP1 - 2.6.32.12-0.7
    root (hd0,0)
    kernel /vmlinuz-2.6.32.12-0.7-default
root=/dev/disk/by-id/scsi-3680fb_06352c1e00015a4dd99a08bbc6e-part2 showopts ide=nodma
apm=off noresume edd=off powersaved=off nohz=off highres=off processor.max_cstate=1
nomodeset x11failsafe vga=0x317
    initrd /initrd-2.6.32.12-0.7-default

待重新启动后执行如下命令：
cat /sys/block/*/queue/scheduler

```

确认除了 none 以外的每一行都有[deadline]，则表示成功修改了启动配置选项，举例如图 6.26 所示。



```

mdw:~ # cat /sys/block/*/queue/scheduler
none
none
none
none
none
none
none
none
none
noop anticipatory [deadline] cfq
noop anticipatory [deadline] cfq

```

图 6.26

(7) 修改/etc/hosts 文件。

/etc/hosts 文件内容举例如下：

```

127.0.0.1    localhost
169.10.35.90 mdw-bmc

```

此文件是为了后面交换 SSH Key。注意此处的名字要和机器实际的 host 名字一致。如果 host 名字在安装操作系统的时候未规划好，则可以通过 hostname 动态修改，同时修改配置文件/etc/sysconfig/network。注意，hostname 命令在 Redhat 和 SUSE 系统中表现不一致，如果是 SUSE 操作系统，则执行命令 hostname+名字，只是临时更改系统的名称，重启后命令失效。

修改/etc/HOSTNAME 里面的主机名，然后执行如下命令：

```

/etc/rc.d/boot.localnet stop
/etc/rc.d/boot.localnet start

```

修改后的主机名就可以生效了。

(8) 配置网络。

采用 `yast2 network` 对每个网卡按照规划的网段和 IP 进行配置, 比较简单的方法是先编辑 `/etc/sysconfig/network/ifcfg-ethx` 文件, 然后执行 `#/etc/init.d/network start(stop)` 命令。

```
etc/init.d # cat /etc/sysconfig/network/ifcfg-eth0
DEVICE=eth7
BOOTPROTO=none
BROADCAST=169.10.35.255
HWADDR=90:e2:ba:13:08:31
IPADDR=169.10.35.48
NETMASK=255.255.255.0
NETWORK=169.10.35.0
ONBOOT=yes
TYPE=Ethernet
MTU=9000
```

可使用 `ethtool eth0` 检查网络通断情况。

RedHat 使用 `mii-tool` 命令检查网络比较方便。RedHat 在修改配置文件时, 注意不要修改 Hardware Address, 以免出现重复或者错误, 导致网卡无法加载。

注意不要配置 IPv6 和 DNS 功能。

(9) 配置时钟同步。

如果用户系统内网有 NTP 服务器, 则可以将所有服务器与 NTP 服务器同步。假设 NTP 服务器的 IP 地址为 169.10.35.161。

需要修改 Master 及所有的 Segment 节点, 具体方法是: 修改 Master 节点的 `/etc/ntp.conf` 文件和备份 Master 节点的 `/etc/ntp.conf` 文件。

将 `server` 修改为:

```
server 169.10.35.161
```

然后执行如下命令:

```
chmod 744 /etc/ntp.conf
mkdir -p /var/lib/ntp/drift/
touch /var/lib/ntp/drift/ntp.drift
chkconfig --level 2345 ntpd on
ntpdate 169.10.35.161
service ntpd start
ntpq -pn
```

NTP 服务器启动后, 需要等待 10 分钟左右才会实现 Segment 同步。

可使用 `date` 命令检查时间。

(10) 创建 Greenplum 用户和组。

在所有机器上创建 Greenplum 用户和组, `root` 和 `gpadmin` 的口令可自由设置, 建议所有机器设置一致。例如:

```
groupadd -g 3030 gpadmin
groupadd -g 3040 gpmon
```

```
useradd -u 3030 -g gpadmin -d /home/gpadmin -s /bin/bash -m gpadmin
useradd -u 3040 -g gpmon -d /home/gpmon -s /bin/bash -m gpmon
passwd gpadmin
```

最后检查设置是否正确，如下：

```
id gpadmin
id gpmon
```

重新启动服务器，使上述所有修改生效。

```
#reboot
```

(11) 安装 Python 2.5。

执行 `/usr/bin/python` 命令检查 Python 版本，使用 `exit()` 方法退出。

只有检查不通过时，才需要安装。

2) 安装 Greenplum (仅需在 Master 节点执行即可)

① 准备文件

创建 `allhosts`、`seghosts` 和 `seginstance` 三个文件，并将其存放到 `/home/gpadmin/` 目录下。其中：

- `allhosts` 文件包含所有主机名称，每个主机名占一行（如机器有多个 IP 则写多个，与 `/etc/hosts` 中的主机名对应），用于交换 SSH。
- `seghosts` 文件仅包含 Segment 的主机名称（如机器有多个 IP，写一个主机名即可），每个主机名占一行，用于批量安装 GP 的 Segment。
- `seginstance` 文件包含所有 Segment instance 的主机名称（一般情况下，一个 IP 对应一个名称），初始化时使用。

② Master 机器安装

以 root 用户登录。

(1) 安装 Greenplum。

复制 `greenplum-db-4.1.1.4-build-1-SuSE10-x86_64.bin` 文件到 `/opt` 目录下。

开始安装：

```
cd opt
chmod 777 -R greenplum-db-4.1.1.4-build-1-SuSE10-x86_64.bin
./greenplum-db-4.1.1.4-build-1-SuSE10-x86_64.bin
```

与路径有关的地方直接按回车键，其他地方均回答 `yes`。

(2) 配置环境变量。

在文件 `/etc/profile` 和 `/home/gpadmin/.bashrc`，`~/.bashrc` 末尾增加如下信息：

```
source /usr/local/greenplum-db/greenplum_path.sh
```

存盘后退出，重新登录。执行 `env` 命令，检查 Greenplum 的环境变量是否生效。

(3) 改变 Greenplum 安装目录的 owner。

```
chown -R gpadmin /usr/local/greenplum-db
chgrp -R gpadmin /usr/local/greenplum-db
chown -R gpadmin /usr/local/greenplum-db-4.1.1.4
```

```
chgrp -R gpadmin /usr/local/greenplum-db-4.1.1.4
```

(4) 创建 Master 主机上的数据存放目录。

```
mkdir /data/master
chown gpadmin /data/master
chgrp gpadmin /data/master
```

③ Segment 机器部署 Greenplum

(1) 准备。

以 root 用户登录 Master 机器，并执行如下命令：

```
source /usr/local/greenplum-db/greenplum_path.sh
```

(2) 改变 root 用户下的 SSH Keys。

```
gpssh-exkeys -f /home/gpadmin/allhosts
```

注：会要求输入每个 Segment 的 root 用户的密码。

如果发现错误，请删除/root/.ssh 下的所有文件，重新执行。

(3) 改变 gpadmin 用户下的 SSH Keys。

```
su - gpadmin
gpssh-exkeys -f /home/gpadmin/allhosts
exit
```

注：会要求输入 gpadmin 用户的密码。

如果发现错误，请删除/home/gpadmin/.ssh 下的所有文件，重新执行。

(4) Segment 安装。

在 Master 机器上使用 root 用户登录，执行如下命令：

```
# gpsegininstall -f /home/gpadmin/seghosts -u gpadmin -p gpadmin
```

其中，-p gpadmin 指的是 gpadmin 用户的密码。

执行以下命令检查：

```
# gpssh -f /home/gpadmin/allhosts -e ll $GPHOME
```

最后执行如下命令：

```
gpssh -f hostfile -e ll $GPHOME
```

如果输出结果显示了 hostfile 中所有相关机器的 \$GPHOME 目录内容，并且属主是 gpadmin，则代表配置成功；如果提示输入密码，则执行如下命令：

```
gpssh -exkeys -f hostfile
```

该命令可以重新同步各台机器的登录证书。

(5) 创建目录及更改 owner。

a. 执行以下命令，连接所有 hosts：

```
# gpssh -f /home/gpadmin/seghosts
```

b. 创建数据目录（所有 Segment 都有效）：

```
mkdir /data/primary/
mkdir /data/mirror
```

c. 改变目录 owner:

```
chown -R gpadmin /data
chgrp -R gpadmin /data
chown -R gpadmin /usr/local/ greenplum-db-4.1.2.1
chgrp -R gpadmin /usr/local/ greenplum-db-4.1.2.1
chown -R gpadmin /usr/local/greenplum-db
chgrp -R gpadmin /usr/local/greenplum-db
```

d. 退出 gpssh:

```
# exit
```

(6) 同步时钟 (可选)。

```
gpssh -f /home/gpadmin/seghosts -v date
gpssh -f /home/gpadmin/seghosts -v ntpd
```

3) 对系统进行验证 (可选)

① 验证操作系统设置

以 gpadmin 用户登录, 然后执行如下命令:

```
source /usr/local/greenplum-db/greenplum_path.sh
```

创建一个文本文件 `hostfile_gpcheck`, 内容是 Greenplum 中涉及机器的主机名称, 每个节点一个名字, 这和安装时的文件是不同的。安装 Greenplum 时的主机配置文件需要写入所有节点的每个网卡 IP 和名字, 本文件要求每个节点一个名字。以实验环境为例:

```
mdw
smdw
sdw1
sdw2
sdw3
```

然后执行如下命令:

```
$ gpcheck -f hostfile_gpcheck -m mdw -s smdw
```

如果没有 Secondary Master 节点, 则去掉 -s。-m 后面是 Master 的 hostname。

验证完成后, 如果操作系统参数配置有问题, 则系统会提示修改哪个参数, 请按照提示进行修改, 然后重新运行该命令即可。

② 验证网络性能

针对 Greenplum 配置中的每个网段 (节点间通信的网段) 编辑一个单独的配置文件, 文件包含该网段涉及的 Segment 主机名称, 然后单独测试每个网段的性能。以实验环境为例, 需要 4 个文件, 如表 6.2 所示。

表 6.2

hostfile1	hostfile2	hostfile3	hostfile4
sdw1-1	sdw1-2	sdw1-3	sdw1-4
sdw2-1	sdw2-2	sdw2-3	sdw2-4
sdw3-1	sdw3-2	sdw3-3	sdw3-4
sdw4-1	sdw4-2	sdw4-3	sdw4-4

对每个网段执行网络性能测试，如下：

```
$ gpcheckperf -f hostfile1 -r N -d /tmp > subnet1.out
$ gpcheckperf -f hostfile2 -r N -d /tmp > subnet2.out
$ gpcheckperf -f hostfile3 -r N -d /tmp > subnet3.out
$ gpcheckperf -f hostfile4 -r N -d /tmp > subnet3.out
```

检查结果是否有问题。如果某个网段速率偏慢，请检查配置情况。

③ 检查存储性能

以 `gpadmin` 用户登录，执行如下命令：

```
$ source /usr/local/greenplum-db/greenplum_path.sh
```

编辑一个文本文件，每个 Segment 的 `hostname` 为一行，实验环境文件为：

```
sdw1
sdw2
sdw3
sdw4
```

然后执行如下命令：

```
$ gpcheckperf -f hostfile_gpcheckperf -r ds -D -d /data/primary -d /data/mirror
```

请确保 `gpadmin` 用户对测试目录有读/写权限。测试需要一段时间，待测试结束后，检查各台机器的读/写速率是否和设计的有差异。如果有问题，请检查 RAID 和磁盘配置。

4) 初始化 Greenplum 数据库系统

仅需在 Master 机器上执行。

① 准备

以 `root` 用户登录，然后执行如下命令：

```
source /usr/local/greenplum-db/greenplum_path.sh
```

② 初始化 GP

(1) 复制初始化模板。

```
cp $GPHOME/docs/cli_help/gpconfigs/gpinitsystem_config /home/gpadmin/gpinitsystem_
config
```

(2) 修改配置文件。

```
chmod 777 /home/gpadmin/gpinitsystem_config
vi /home/gpadmin/gpinitsystem_config
```

举例如下：

```
ARRAY_NAME="Greenplum"
SEG_PREFIX=gp
PORT_BASE=50000
declare -a DATA_DIRECTORY= (/data/primary /data/primary /data/primary)
declare -a MIRROR_DATA_DIRECTORY=(/data/mirror /data/mirror /data/mirror)
MASTER_HOSTNAME=mdw #机器名
MASTER_DIRECTORY=/data/master
MASTER_PORT=5432
MACHINE_LIST_FILE=/home/gpadmin/seginstance
```

(3) 执行初始化命令。

以 `gpadmin` 用户登录，然后执行如下命令：

```
su gpadmin
gpinitssystem -c /home/gpadmin/gpinitssystem_config -S
```

如果安装过程出错，则系统会在用户的 `home` 目录准备一个回滚脚本：

```
~/gpAdminLogs/backout_gpinitssystem_<user>_<timestamp>
```

直接运行该脚本，然后重新初始化即可。

(4) 设置系统环境变量。

编辑 `/home/gpadmin/.bashrc` 文件，在文件末尾添加如下内容：

```
source /usr/local/greenplum-db/greenplum_path.sh
export MASTER_DATA_DIRECTORY=/data/master/gpseg-1
```

文件存盘退出并执行如下命令：

```
source /home/gpadmin/.bashrc
```

③ 创建数据库

以 `gpadmin` 用户登录（`su gpadmin`），执行如下命令：

```
psql postgres
```

然后执行如下命令：

```
create database nethouse;
```

执行以下脚本退出：

```
\q
```

④ 允许客户端连接

修改配置文件 `/data/master/gpseg-1/pg_hba.conf`，在最后添

加一行（最后一行）：

```
host all gpadmin 169.10.35.91/32 trust
host all gpadmin 192.168.50.91/32 trust
host all gpadmin 192.168.80.91/32 trust
host all gpadmin 0.0.0.0/0 trust
```

⑤ pgAdmin 连接方法

参数配置如图 6.27 所示。

可以看出，IP 地址是 Master 的地址，端口号是 5432。

⑥ 修改数据库配置文件

代码如下：

```
su - gpadmin
```

```
gpconfig -c max_statement_mem          -v 1024MB    -m 4096MB
gpconfig -c statement_mem              -v 512MB     -m 512MB
gpconfig -c gp_vmem_protect_limit      -v 12288    -m 12288
gpconfig -c max_appendonly_tables     -v 4096      -m 4096
gpconfig -c max_fsm_relations          -v 5000      -m 5000
```

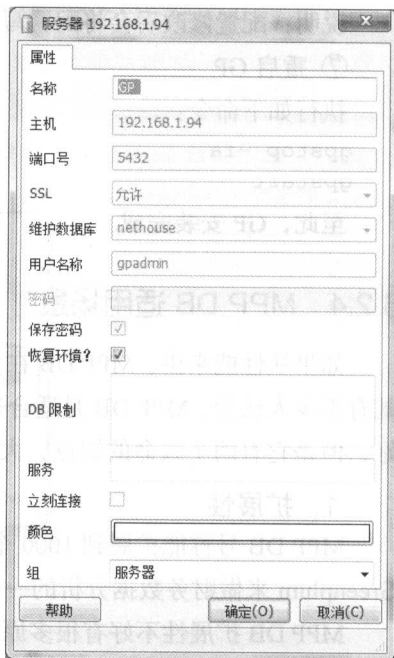


图 6.27

```

gpconfig -c default_tablespace          -v ts_cfg      -m ts_cfg
gpconfig -c log_duration                 -v on           -m on
gpconfig -c log_min_messages             -v WARNING     -m WARNING
gpconfig -c gp_max_filesapces            -v 50          -m 50
gpconfig -c gp_max_tablesapces           -v 50          -m 50
gpconfig -c gp_segment_connect_timeout   -v 600s        -m 600s
gpconfig -c gp_interconnect_setup_timeout -v 600s        -m 600s
gpconfig -c max_connections              -v 150         -m 100
gpconfig -c max_prepared_transactions     -v 200         -m 200
gpconfig -c gp_external_max_segs         -v 20          -m 20
gpconfig -c gp_resqueue_memory_policy     -v auto        -m auto

gpconfig -c max_connections              -v 800         -m 100
gpconfig -c max_work_mem                 -v 1024MB      -m 1024MB
gpconfig -c work_mem                     -v 512MB       -m 512MB

```

检查是否生效（需要重启才能生效），执行如下命令：

```

psql nethouse0
nethouse0=# show statement_mem

```

注意其中的 `nethouse0` 是数据库，请根据实际情况填写。

说明：配置修改后会将配置写入文件 `/data/master/gpseg-1/postgresql.conf` 中。

⑦ 重启 GP

执行如下命令：

```

gpstop -fa
gpstart

```

至此，GP 安装完成。

6.2.4 MPP DB 适用场景^①

如果从性能来讲，MPP DB 在多维复杂查询方面的性能确实要好于 Hive、HBase、Impala 等，因此有不少人认为，MPP DB 是适合这种场景的未来解决方案。MPP DB 看似对多维度复杂查询性能较好，但是它有两个致命的缺点，大家在选型的时候不得不考虑。

1. 扩展性

MPP DB 号称能扩展到 1000 个节点以上，但在实际应用中不超过 100 个节点，如在支付宝中用 Greenplum 来做财务数据分析的一个最大集群只有 60 多台机器。

MPP DB 扩展性不好有很多原因，最根本的原因是架构本身。MPP DB 是基于原 DB 扩展而来的，DB 中天然追求一致性（Consistency），必然会带来分区容错性较差。当集群规模变得太大、业务数据太多时，MPP DB 的元数据管理就完全是一个灾难。元数据巨大无比，一旦出错，将很难恢复。

所以 MPP DB 要在扩展性上有质的提升，就要对元数据及数据存储有架构上的突破，降低对一致性的要求，否则很难相信一个 MPP DB 数据库是易于扩展的。

^① MPP DB 是大数据实时分析系统未来的选择吗，<http://jiezhuzhu2007.iteye.com/blog/2064588>。

2. 并发的支持

一个查询系统设计出来就是供用户使用的，所以能支持的并发数越多越好。MPP DB 的核心原理是将一个大的查询通过分解为一个个子查询，分布到底层执行，最后再合并结果，也就是通过多线程并发来暴力扫描以实现高速。这种暴力扫描的方法对单个查询来说动用了整个系统的能力，所以单个查询比较快，但同时带来用力过猛的问题，整个系统能支持的并发数必然不多。从目前的实际经验来看，也就支持 50~100 的并发能力。

当前 HBase 和 Impala 在应对复杂查询时，也是通过全盘扫描的方法来实现的，在这种场景下，硬盘数量越多越好，转速越快越好。HBase 为什么号称支持上千并发，这也是在特定的场景下（查询时带用户标识，即带 row key）才能实现的。在复杂的查询场景下，任何系统都将崩溃。

所以，MPP DB 目前更适合小集群（100 以内）、低并发（50 左右）的场景。

6.3 SQL on Hadoop

SQL on Hadoop 是一个泛化的概念，是指 Hadoop 领域里一系列支持 SQL 接口的组件和技术。下面讨论几种常见的 SQL on Hadoop 技术。

6.3.1 Hive

Hive 的基本架构如图 6.28 所示。

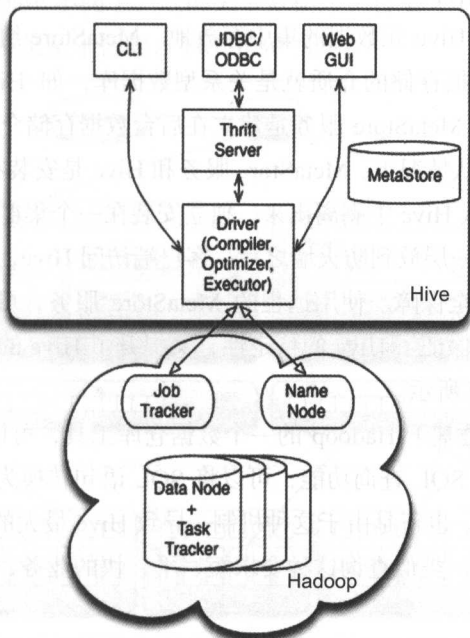


图 6.28

由图 6.28 可知，Hadoop 和 MapReduce 是 Hive 架构的根基。Hive 架构包括如下组件：CLI（Command Line Interface）、JDBC/ODBC、Thrift Server、Web GUI、MetaStore 和 Driver（Complier、Optimizer 和 Executor）。这些组件可以分为两大类：服务端组件和客户端组件。

首先介绍一下服务端组件。

Driver 组件：该组件包括 Complier、Optimizer 和 Executor，其作用是将 HiveQL（类 SQL）语句进行解析、编译优化，生成执行计划，然后调用底层的 MapReduce 计算框架。

MetaStore 组件：元数据服务组件，这个组件存储 Hive 的元数据。Hive 的元数据存储存储在关系型数据库里，Hive 支持的关系型数据库有 Derby、MySQL。元数据对于 Hive 来说十分重要，因此 Hive 支持把 MetaStore 服务独立出来，安装到远程的服务器集群里，从而解耦 Hive 服务和 MetaStore 服务，保证 Hive 运行的健壮性。

Thrift 服务：Thrift 是 Facebook 开发的一个软件框架，用来进行可扩展且跨语言的服务的开发。Hive 集成了该服务，能让不同的编程语言调用 Hive 的接口。

其次介绍一下客户端组件。

CLI：Command Line Interface，命令行接口。

Thrift 客户端：图 6.28 所示的架构图里没有写上 Thrift 客户端，但是 Hive 架构的许多客户端接口都是建立在 Thrift 客户端之上的，包括 JDBC 和 ODBC 接口。

Web GUI：Hive 客户端提供了一种通过网页的方式访问 Hive 所提供的服务。这个接口对应 Hive 的 HWI（Hive Web Interface）组件，使用前要启动 HWI 服务。

下面着重介绍 MetaStore 组件。

Hive 的 MetaStore 组件是 Hive 元数据的集中存放地。MetaStore 组件包括两部分：MetaStore 服务和后台数据的存储。后台数据存储的介质就是关系型数据库，如 Hive 默认的嵌入式磁盘数据库 Derby，以及 MySQL 数据库。MetaStore 服务是建立在后台数据存储介质之上，并且可以和 Hive 服务进行交互的服务组件。在默认情况下，MetaStore 服务和 Hive 是安装在一起的，运行在同一个进程中。也可以把 MetaStore 服务从 Hive 中剥离出来，独立安装在一个集群里，Hive 远程调用 MetaStore 服务，这样就可以把元数据这一层放到防火墙之后，客户端访问 Hive，就可以连接到元数据这一层，从而提供了更好的管理性和安全保障。使用远程的 MetaStore 服务，可以让 MetaStore 服务和 Hive 运行在不同的进程里，这样既保证了 Hive 的稳定性，又提升了 Hive 的效率。

Hive 的执行流程如图 6.29 所示。

一句话描述 Hive：Hive 是基于 Hadoop 的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供完整的 SQL 查询功能，可以将 SQL 语句转换为 MapReduce 任务运行。Hive 支持 HSQL，这是一种类 SQL。也正是由于这种机制，导致 Hive 最大的缺点是慢。Map/Reduce 调度本身只适合批量、长周期任务，类似查询这种要求短、平、快的业务，代价太高。

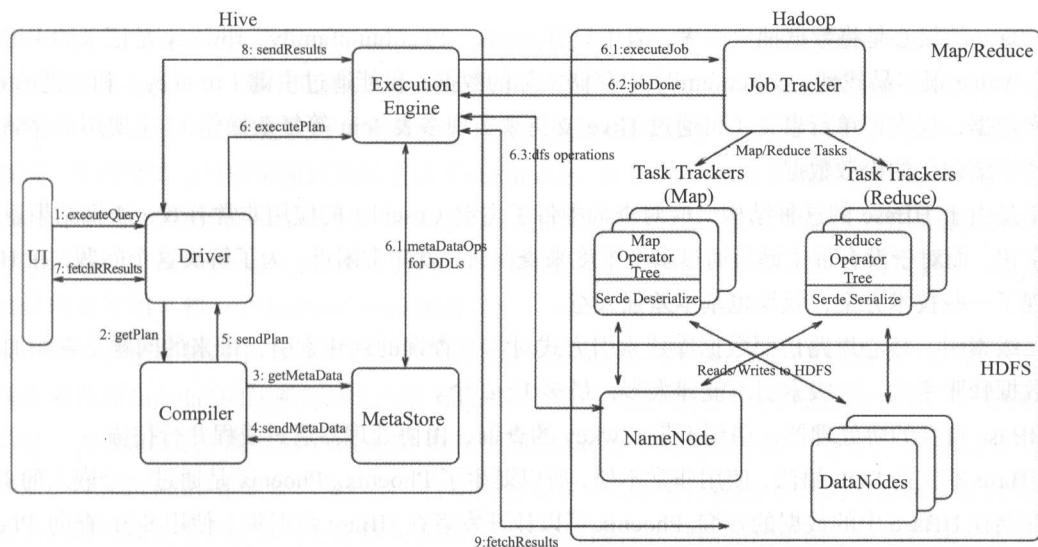


图 6.29

6.3.2 Phoenix

HBase 是一个分布式的、面向列的开源数据库，该技术来源于 Changetal 所撰写的 Google 论文“Bigtable: 一个结构化数据的分布式存储系统”。就像 Bigtable 利用了 Google 文件系统 (File System) 所提供的分布式数据存储一样，HBase 在 Hadoop 之上提供了类似 Bigtable 的能力。HBase 是 Apache 的 Hadoop 项目的子项目。HBase 不同于一般的关系数据库，它是一个适合非结构化数据存储的数据库，而且 HBase 是基于列而不是基于行的，如图 6.30 所示。

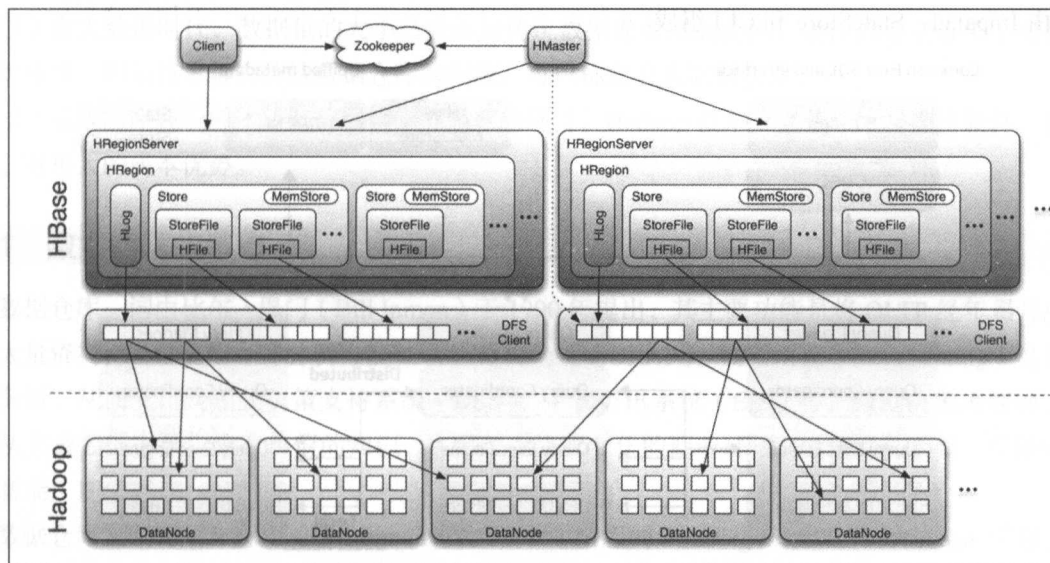


图 6.30

HBase 的核心是将数据抽象成表，表中只有 rowkey 和 columnfamily。rowkey 是记录的主键，通过 Key/Value 很容易找到；columnfamily 中存储实际的数据。仅能通过主键（rowkey）和主键的 range 来检索数据，仅支持单行事务（可通过 Hive 支持来实现多表 Join 等复杂操作）。主要用来存储非结构化和半结构化的松散数据。

正是由于 HBase 的这种结构，应对查询中带了主键（userid）的应用非常有效，查询结果返回速度非常快。而对于没有带主键且通过多个维度来查询时，就非常困难。为了解决这个问题，在 HBase 上实现了一些技术方案，效果也基本差强人意。

二级索引，核心思路仿照数据库建索引方式对需要查询的列建索引，带来的问题是影响加载速度、数据膨胀率大。二级索引不能建太多，最多 1~2 个。

HBase 自身的协处理器，遇到不带 rowkey 的查询，由协处理器通过线程并行扫描。

HBase 不支持 SQL 语法，使用非常不便，所以诞生了 Phoenix。Phoenix 是通过一个嵌入的 JDBC 驱动存储在 HBase 中的数据的查询。Phoenix 可以让开发者在 HBase 数据集上使用 SQL 查询。Phoenix 查询引擎会将 SQL 查询转换为一个或多个 HBase 的扫描操作，并合并执行以生成标准的 JDBC 结果集。对于简单查询来说，Phoenix 的性能甚至胜过 Hive。

6.3.3 Impala

Impala 是 Cloudera 在受到 Google 的 Dremel 启发下开发的实时交互 SQL 大数据查询工具。Impala 没有再使用缓慢的 Hive+MapReduce 批处理，在架构上使用了与传统并行关系数据库中类似的分布式查询引擎（由 QueryPlanner、QueryCoordinator 和 QueryExecEngine 三部分组成），可以直接从 HDFS 或 HBase 中用 SELECT、JOIN 和统计函数查询数据，从而大大降低了延迟。其架构如图 6.31 所示，主要由 Impalad、StateStore 和 CLI 组成。

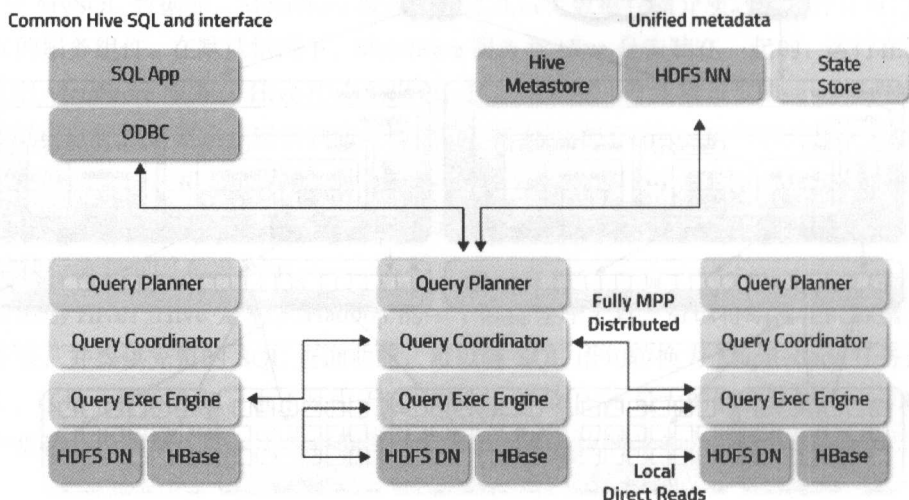


图 6.31

Impalad: 与 DataNode 运行在同一节点上, 由 Impalad 进程表示, 它接收客户端的查询请求 (接收查询请求的 Impalad 为 Coordinator, Coordinator 通过 JNI 调用 Java 前端解释 SQL 查询语句, 生成查询计划树, 再通过调度器把执行计划分发给具有相应数据的其他 Impalad 执行), 读/写数据, 并行执行查询, 并把结果通过网络流式地传送给 Coordinator, 由 Coordinator 返回给客户端。同时 Impalad 也与 StateStore 保持连接, 用于确定哪个 Impalad 是健康的, 可以接受新的工作。在 Impalad 中启动三个 ThriftServer——beeswax_server (连接客户端)、hs2_server (借用 Hive 元数据)、be_server (Impalad 内部使用) 和一个 ImpalaServer 服务。

StateStore: 跟踪集群中 Impalad 的健康状态及位置信息, 由 StateStored 进程表示, 它通过创建多个线程来处理 Impalad 的注册订阅和与各 Impalad 保持心跳连接, 各 Impalad 都会缓存一份 StateStore 中的信息, 当 StateStore 离线后 (Impalad 发现 StateStore 处于离线状态时, 会进入 Recovery 模式, 反复注册, 当 StateStore 重新加入集群后, 自动恢复正常, 更新缓存数据), 因为 Impalad 有 StateStore 的缓存, 仍然可以工作, 但会因为有些 Impalad 失效且已缓存的数据无法更新, 从而把执行计划分配给失效的 Impalad, 导致查询失败。

CLI: 提供给用户查询使用的命令行工具 (Impala Shell 使用 Python 实现), 同时还提供了 Hue、JDBC、ODBC 等使用接口。

6.4 大数据仓库

前面讲到的 MPP DB、SQL on Hadoop 实际解决的都是传统数据仓库的多维查询问题, 但为什么大家不用数据仓库去解决呢? 核心原因有两个:

(1) 在大数据时代, 数据量的大小已经远超传统数据库处理的范围, 数据的量变带来了技术需求的质变。用新技术解决传统数据仓库的问题, 可以称之为大数据仓库。

(2) 成本的原因。相比传统数据仓库的高性能硬件, Hadoop 技术一般使用大量廉价硬件, 相对而言, 有很大的成本优势。

6.4.1 数据仓库的概念

数据仓库一词由比尔·恩门 (Bill Inmon) 于 1990 年提出, 其主要功能是将 OLTP 经年累月所积累的大量资料, 通过使用各种分析方法如联机分析处理 (OLAP)、数据挖掘 (Data Mining) 进行系统的分析, 从而支持构建如决策支持系统 (DSS)、主管资讯系统 (EIS) 等, 帮助决策者快速、有效地从大量数据中分析出有价值的知识, 从而对应快速变化的商业环境, 做出最佳决策, 帮助建构商业智能 (BI)。

数据仓库之父比尔·恩门 (Bill Inmon) 在 1991 年出版的 *Building the Data Warehouse* (《建立数据仓库》) 一书中所提出的定义被广泛接受: 数据仓库 (Data Warehouse) 是一个面向主题的 (Subject

Oriented)、集成的 (Integrated)、相对稳定的 (Non-Volatile)、反映历史变化 (Time Variant) 的数据集合，用于支持管理决策 (Decision Making Support)。

6.4.2 OLTP/OLAP 对比

数据仓库里面有 OLTP/OLAP 之分，OLTP 是传统关系型数据库的主要应用，其主要面向基本的、日常的事务处理，如银行交易；OLAP 是数据仓库系统的主要应用，支持复杂的分析操作，侧重决策支持，并且提供直观易懂的查询结果，如图 6.32 所示。

	OLTP	OLAP
面向应用	日常交易处理	明细查询，分析决策
访问模式	简单小事务，操作少量数据	复杂聚合查询，可以操作大量数据
数据	当前最新数据	历史数据
数据规模	GB	TB ~ PB
数据更新	实时更新	批量更新
数据组织	满足3NF	反范式，星形模型

图 6.32

6.4.3 大数据场景下的同与不同

在大数据时代，大数据仓库面对的最基本、最典型的场景还是传统的 OLAP 场景，最明显的区别是数据规模的急剧膨胀，从传统的单表千万级到现在的单表百亿、万亿级，维度也从传统的几十维到现在的一些互联网企业可能存在的万维。因为系统的交互对象是人，虽然数据量急剧增大，但系统的响应延迟要求仍为秒级。图 6.33 是阿里 ADS 对当前业界一些常见的分析仓库从支持规模和响应时间上的分类，有一定的参考意义。

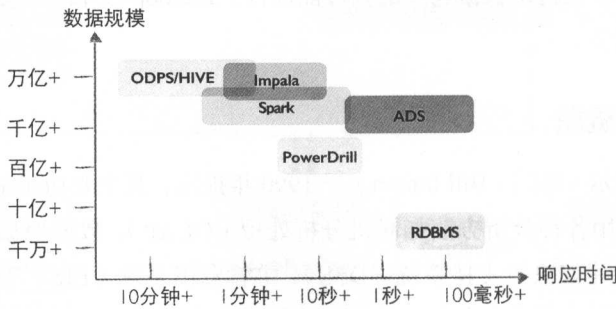


图 6.33

在大数据时代，数据价值越来越大，分析手段和分析工具也越来越多。传统 SQL 包打天下的局面可能不复存在，SQL、Python、R、BI 工具/可视化工具都有需求。因此，除了性能之外，大数据仓库必然支持更多的处理模式和接口，通常称之为“泛 SQL”。

要解决传统数据仓库不能解决的扩展性和性能问题，需要从存储引擎和查询引擎两个层次进行优化。下面讲解一些常见的思路和技术。

6.4.4 查询引擎

传统数据库为了追求性能，在设计的时候将存储引擎和查询引擎耦合在一起，从而带来扩展性不佳的问题。大数据仓库的一个明显的设计思路是对存储引擎和查询引擎分别进行扩展和优化。

目前，大数据仓库优化在技术上和传统的数据库相比并没有质的突破，优化器主要有基于规则的优化器（RBO）与基于代价的优化器（CBO）两种。

下面介绍一种基于统计的优化技术——直方图技术^①。

1. 何谓直方图

在分析表或索引时，直方图用于记录数据的分布。通过获得该信息，基于成本的优化器就可以决定使用返回少量行的索引，而避免使用基于限制条件返回许多行的索引。直方图的使用不受索引的限制，可以在表的任何列上构建直方图。

直方图类似图 6.34。可以看到，一张表只有两个字段（Start_time 和 User_ID）和 3 个分区。创建直方图就是对列上的信息进行统计，如选取 U1/U2/U3，构造出来的统计信息类似直方图。

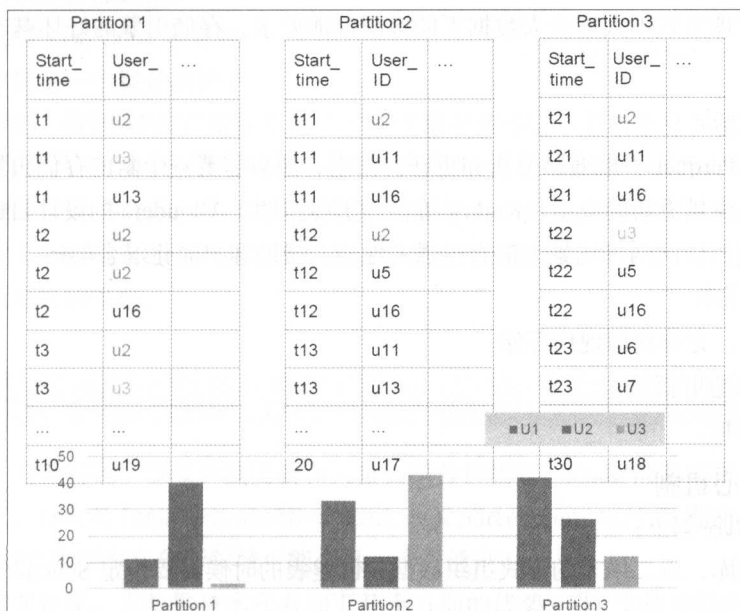


图 6.34

构造直方图是为了帮助优化器在表中数据严重偏斜时做出更好的规划。例如，如果 1~2 个值构成了表中的大部分数据（数据偏斜），相关的索引就可能无法帮助减少满足查询所需的 I/O 数量。创

^① 直方图技术，<http://www.oraclema.com/oracle/oracle-histograms.html>。

建直方图可以让基于成本的优化器知道何时使用索引最合适，或者何时根据 WHERE 子句中的值返回表中 80% 的记录。

2. 何时使用直方图

通常情况下，建议在以下场合中使用直方图：

- (1) 当 WHERE 子句引用了列值分布存在明显偏差的列时。当这种偏差相当明显，以至于 WHERE 子句中的值会使优化器选择不同的执行计划时，应该使用直方图来帮助优化器修正执行路径。
- (2) 当列值导致不正确的判断时。这种情况通常会发生的多表连接时。

3. 直方图的种类

Oracle 利用直方图来提高非均匀数据分布的选择率和技术的计算精度。但是，实际上，Oracle 会采用两种不同的策略来生成直方图：一种是针对包含很少不同值的数据集；另一种是针对包含很多不同值的数据集。Oracle 会针对第一种情况生成频率直方图，针对第二种情况生成高度均衡直方图。通常情况下，当 BUCKET 小于表的 NUM_DISTINCT 值时，得到的是高度均衡直方图；而当 BUCKET 大于表的 NUM_DISTINCT 值时，得到的是频率直方图。

6.4.5 存储引擎

传统的数据管理方式不太适合大数据下的极限性能要求，存储引擎的好坏基本决定了整个数据仓库的基础。

1. Kudu 目标

由于 HBase、Parquet 不能兼顾分析和更新的需求，所以需要一个新的存储引擎可以同时支持高吞吐的分析应用及少量更新的应用，Kudu 存储引擎应运而生。Cloudera 的设计目标如下^①：

- 在扫描和随机访问两种场景下都有很强的性能，帮助客户简化混合架构。
- 高 CPU 利用率。
- 高 I/O 效率，充分利用现代存储。
- 支持数据原地更新。
- 支持双活集群。

2. Kudu 核心机制

Kudu 的核心机制如下：

- (1) 模仿数据库，以二维表的形式组织数据，创建表的时候需要指定 Schema，所以只支持结构化数据。
- (2) 每张表指定一个或多个主键。
- (3) 支持 insert、update、delete 操作，这些修改操作都要指定主键。
- (4) read 操作，只支持 scan 原语。

^① Cloudera 的设计目标，<http://blog.cloudera.com/blog/2015/09/kudu-new-apache-hadoop-storage-for-fast-analytics-on-fast-data/>。

- (5) 一致性模型, 默认支持 snapshot。
- (6) Cluster 类似 HBase 简单的主备结构。
- (7) 单张表支持水平分割。
- (8) 使用 Raft 协议, 可以根据 SLA 指定备份块数量。
- (9) 列式存储。
- (10) 数据先更新到内存中, 最后再合并到最终存储中, 有专门的后台进程负责。
- (11) 延迟物化, 好处是代码实际执行时, 对于一些选择性条件的查询, 可以直接跳过不必要的数
据。
- (12) 支持和 MR、Spark、Impala 等集成, 支持 Locality、Columnar Projection、Predicate
Pushdown 等。

3. Mesa 技术

传统的数据仓库通常会遇到两个问题: (1) 更新的 throughput 不高; (2) 更新影响查询。

为了解决这两个问题, Google 的 Mesa 系统设计了一个 MVCC 数据模型, 通过增量更新和合并技术, 将离散的更新 I/O 转变成批量 I/O, 平衡了查询和更新的冲突, 提高了更新的吞吐量。

Mesa 设计了一种多版本管理技术来解决更新的问题:

- (1) 使用二维表来管理数据。
- (2) 每个字段用 Key/Value 来管理。
- (3) 每个字段指定一个聚合函数 F。
- (4) 数据更新时, 按照 MVCC 增量更新, 并给增量更新指定一个版本号 N 和谓词 P。
- (5) 在进行查询时, 自动识别聚合函数, 把所有版本的更新按照聚合函数自动计算出来。
- (6) 多版本如果永远不合并, 那么存储的代价将非常大, 而且因为每次查询都需要遍历所有版本
号, 所以版本过多会影响查询, 因此, 定期的合并是自然而然的。
- (7) 采用两段更新的策略。

6.5 小结

为了应对 BI (商业智能) 领域少量更新和大量扫描分析场景, Kudu 借鉴了很多传统的数据仓库技术。在这个领域, 目前是 Impala+Kudu/Hive/Spark SQL/Greenplum MPP 数据库在混战, 未来将走向融合, 传统的 MPP 数据库在分析领域可能会是一个过渡产品。

相比传统的数据仓库, 大数据技术有如下几个方面的优势: ① 支持非结构化数据; ② 扩展性增强; ③ 与新的分析方法和算法的结合; ④ 成本降低。

相比传统数据仓库, 大数据也有很多劣势: ① 小数据量时比传统的 MPP 差, 大数据量时不能满足交互式分析秒级响应的需求; ② 对 SQL 的支持不充分等。所以业界有不少厂商在做这方面的探索, 如 Cloudera 的 Impala、星环的 Inceptor、阿里的 ADS、百度的 Palo 等。

第 7 章

批处理技术

定义：复杂的批量数据处理（Batch Data Processing），通常的时间跨度在几分钟到数小时之间。

7.1 批处理技术的概念

数据批处理，发展最早，应用也最为广泛。其最主要的应用场景就是传统的 ETL 过程，如电信领域的 KPI、KQI 计算。单据经过探针采集上来后，按照一定的规则转换成原始单据，根据业务需求，按周期（15 分钟、60 分钟、天）等粒度计算成业务单据。这一过程使用数据库来承担，传统的数据库扩展性遇到瓶颈后，就出现了 MPP 技术。Google 的研究员另辟蹊径，从传统的函数式编程里得到灵感，发明了 MapReduce，使得大规模扩展成为可能。Spark 一开始是为了替代 MapReduce，后来逐渐发展成数据处理统一平台。除了迭代式的计算外，大规模机器学习需要另外的框架，所以本章还会讲到 BSP 技术。在这个过程中会讲到两种关键技术：一种是 CodeGen，另一种是 CPU 亲和技术。批处理为了提高吞吐量，CPU 的利用率是关键。

7.2 MPP DB 技术

传统的数据库人员第一时间想到的是用数据库来承担 ETL 的后分析，因为数据库的最大好处是对 SQL 的支持。

第 6 章详细介绍了以 Greenplum 为代表的 MPP 数据库，这类数据库突破了传统数据库单点的瓶颈，扩展性得到一定的提升，在一定规模的数据量下（通常是 TB 级的数据处理），MPP 可以起到很好的效果。在数据量持续上升的情况下（PB 级以上），MPP 由于自身架构上的限制，遇到了明显的扩展瓶颈。Hadoop 的出现，解决了扩展性问题。

另外，MPP 的计算和存储过程是耦合的，这方面比不上 MapReduce、HDFS 的分离设计。分离设计的最大优点是，除了 MapReduce 引擎外，还可以根据业务需求选择图计算、深度学习等其他框架。从目前的应用来说，一份数据选择多个引擎以应对多个业务是必然的选择。

7.3 MapReduce 编程框架

7.3.1 MapReduce 起源

Hadoop 的思想来源于 Google 的几篇论文，论文中写道，MapReduce 的灵感来源于函数式语言（如 Lisp）中的内置函数 Map 和 Reduce。简单来说，在函数式语言里，Map 表示对一张列表（List）中的每个元素进行计算，Reduce 表示对一张列表中的每个元素进行迭代计算。它们具体的计算是通过传入的函数来实现的，而 Map 和 Reduce 提供的是计算的框架。不过，从这样的解释到现实中的 MapReduce 相差太远，仍然需要一个跳跃。再仔细看，Reduce 既然能做迭代计算，那就表示列表中的元素是相关的；而 Map 则对列表中的每个元素做单独处理，这表示列表中的数据是杂乱无章的。这样看来，就有点联系了。在 MapReduce 里，Map 处理的是原始数据，自然是杂乱无章的，各条数据之间没有联系；到了 Reduce 阶段，数据是以 Key 后面跟着若干个 Value 来组织的，这些 Value 有相关性，符合函数式语言里 Map 和 Reduce 的基本思想。

这样就可以把 MapReduce 理解为：把一堆杂乱无章的数据按照某种特征归纳起来，然后处理并得到最后的结果。Map 面对的是杂乱无章的、互不相关的数据，它解析每个数据，从中提取出 Key 和 Value，也就是提取数据的特征。经过 MapReduce 的 Shuffle 阶段之后，在 Reduce 阶段看到的是已经归纳好的数据，在此基础上可以做进一步处理以便得到结果。

7.3.2 MapReduce 原理

MapReduce 是一种云计算的核心计算模式，是一种分布式运算技术，也是简化的分布式并行编程模式，主要用于大规模并行程序并行问题。

MapReduce 模式的主要思想是自动将一个大的计算（如程序）拆解成 Map（映射）和 Reduce（化简）的方式，流程图如图 7.1 所示。

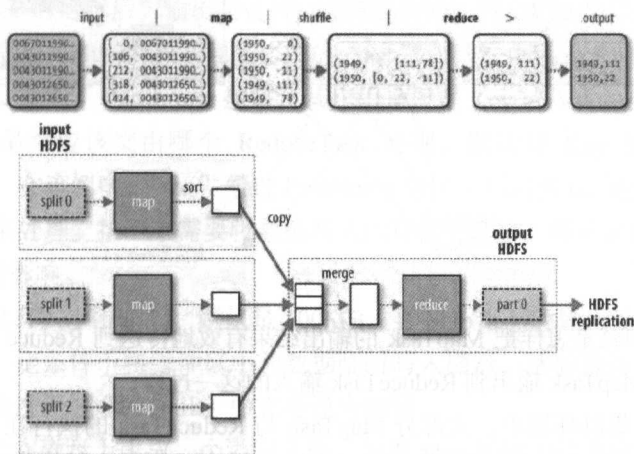


图 7.1

数据被分割后，通过 Map 函数将数据映射成不同的区块，分配给计算机集群进行处理，以达到分布式运算的效果，再通过 Reduce 函数将结果汇整，从而输出开发者所需的结果。

MapReduce 借鉴了函数式程序设计语言的设计思想，其软件实现是指定一个 Map 函数，把键值对 (Key/Value) 映射成新的键值对 (Key/Value)，形成一系列中间结果形式的键值对 (Key/Value)，然后把它们传递给 Reduce (规约) 函数，把具有相同中间形式 Key 的 Value 合并在一起。Map 和 Reduce 函数具有一定的关联性。

MapReduce 致力于解决大规模数据处理的问题，因此在设计之初就考虑了数据的局部性原理，将整个问题分而治之。MapReduce 集群由普通 PC 构成，为无共享式架构。在处理之前，将数据集分布至各个节点；在处理时，每个节点就近读取本地存储的数据处理 (Map)，将处理后的数据进行合并 (Combine)、排序 (Shuffle and Sort) 后再分发 (至 Reduce 节点)，从而避免了大量数据的传输，提高了处理效率。无共享式架构的另一个好处是配合复制 (Replication) 策略，集群可以具有良好的容错性，一部分节点宕机不会影响整个集群的正常工作。

7.3.3 Shuffle

Shuffle 过程是 MapReduce 的核心，也被称为奇迹发生的地方。Shuffle 的原意是洗牌或弄乱，可能大家更熟悉的是 Java API 里的 Collections.shuffle(list) 方法，它会随机地打乱参数 list 里的元素顺序。如果读者不知道 MapReduce 里的 Shuffle 是什么，那么请看图 7.2。

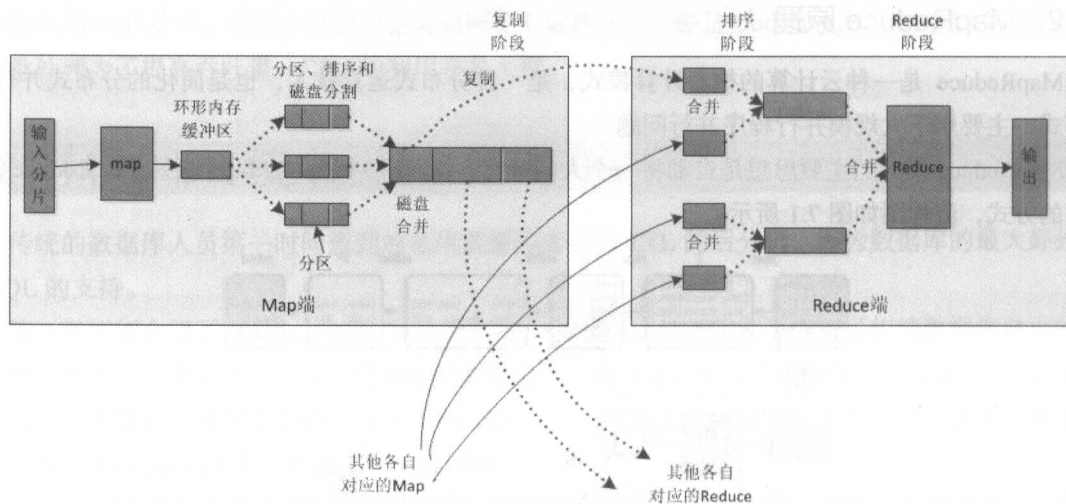


图 7.2

Shuffle 的大致范围就是怎样把 MapTask 的输出结果有效地传送到 Reduce 端。也可以这样理解，Shuffle 描述了数据从 MapTask 输出到 ReduceTask 输入的这一过程。

在 Hadoop 这样的集群环境中，大部分 MapTask 与 ReduceTask 的执行在不同的节点上。当然，很多情况下 Reduce 在执行时需要跨节点去拉取其他节点上的 MapTask 结果。如果集群正在运行的 Job

有很多，那么 Task 的正常执行对集群内部的网络资源消耗会很严重。这种网络消耗是正常的，我们不能限制，能做的就是最大化地减少不必要的消耗。另外，在节点内，相比于内存，磁盘 I/O 对 Job 完成时间的影响也是可观的。从最基本的要求来说，我们对 Shuffle 过程的期望可以有：完整地拉取 Map 端数据到 Reduce 端；在跨节点拉取数据时，尽可能地减少对带宽的不必要消耗；减少磁盘 I/O 对 Task 执行的影响。以 WordCount 为例，假设它有 8 个 MapTask 和 3 个 ReduceTask。从图 7.2 中可以看出，Shuffle 过程横跨 Map 与 Reduce 两端，所以接下来也会分两部分来讲解。先看看 Map 端的情况，如图 7.3 所示。

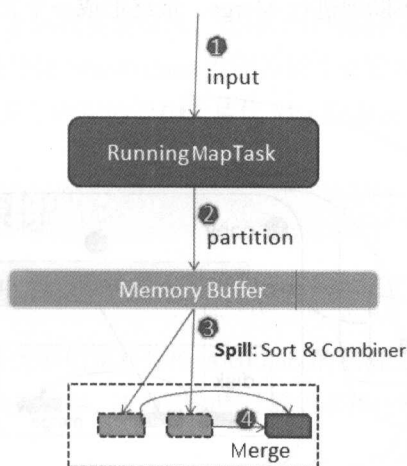


图 7.3

整个流程分为四步。

(1) 在 MapTask 执行时，其输入数据来源于 HDFS 的 Block。Split 与 Block 的对应关系默认是一一对一。在 WordCount 例子中，假设 Map 的输入数据都是像“aaa”这样的字符串。

(2) 在经过 Mapper 的运行后，输出是这样一个 Key/Value 对：Key 是“aaa”，Value 是数值 1。我们知道这个 Job 有 3 个 ReduceTask，到底当前的“aaa”应该交由哪个 Reducer 去处理，是需要现在决定的。MapReduce 提供了 Partitioner 接口，其作用是根据 Key 或 Value 及 Reduce 的数量来决定当前的这对输出数据最终应该交由哪个 ReduceTask 处理。默认对 Key 进行哈希运算后，再以 ReduceTask 数量取模。在该例中，“aaa”经过 Partition (分区) 后返回 0，也就是这对输出数据应当交由第一个 Reducer 来处理。接下来需要将数据写入内存缓冲区中。缓冲区的作用是批量收集 Map 结果，减少磁盘 I/O 的影响。

(3) 内存缓冲区是有大小限制的，默认是 100MB。当 MapTask 的输出结果有很多时，内存可能会不足，所以需要在一定条件下将缓冲区中的数据临时写入磁盘，然后重新利用这个缓冲区。这个从内存往磁盘写数据的过程被称为 Spill，中文可译为溢写。

(4) 每次溢写都会生成一个溢写文件，如果 Map 的输出结果很大，就会有多次这样的

溢写发生，磁盘上就会有多个溢写文件存在。当 MapTask 真正完成时，内存缓冲区中的数据将全部溢写到磁盘中形成一个溢写文件。最终磁盘上至少会有一个这样的溢写文件存在（如果 Map 的输出结果很少，那么当 Map 执行完成时，只会产生一个溢写文件）。因为最终的文件只有一个，所以需要将这溢写文件归并到一起，这个过程就叫作 Merge。至此，Map 端的所有工作都已结束。

每个 ReduceTask 不断地通过 RPC 从 JobTracker 那里获取 MapTask 是否完成的信息。如果 ReduceTask 获知某台 TaskTracker 上的 MapTask 执行完成，那么 Shuffle 的后半段过程开始启动。简单地说，ReduceTask 在执行之前工作就是不断地拉取当前 Job 里每个 MapTask 的最终结果，然后对从不同地方拉取过来的数据不断地进行 Merge，最终形成一个文件作为 ReduceTask 的输入文件，如图 7.4 所示。

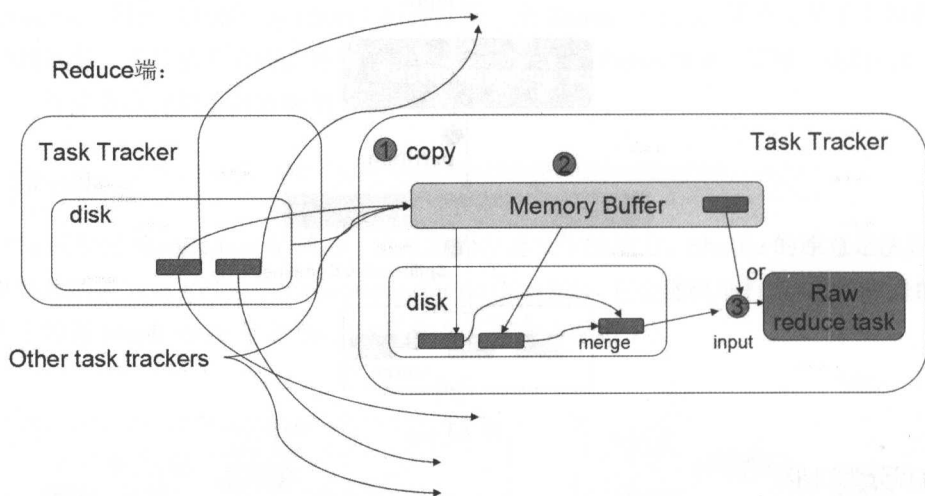


图 7.4

如同 Map 端的细节图，Shuffle 在 Reduce 端的过程也能用图上标明的三点来概括。

(1) copy 过程，即简单地拉取数据。Reduce 进程启动一些数据 copy 线程 (Fetcher)，通过 HTTP 方式请求 MapTask 所在的 TaskTracker 获取 MapTask 的输出文件。因为 MapTask 早已结束，所以这些文件就由 TaskTracker 管理。

(2) Merge 阶段。同 Map 端的 Merge 动作，只是数组中存放的是不同 Map 端复制过来的数据。复制过来的数据会先放入内存缓冲区中，当内存中的数据量到达一定阈值时，就会启动内存到磁盘的 Merge。与 Map 端类似，这也是溢写的过程，会在磁盘中生成众多的溢写文件，然后将这些溢写文件进行归并。

(3) Reducer 的输入文件。不断进行 Merge 后，最后会生成一个“最终文件”。这个文件可能存放在磁盘上，也可能存放在内存中，默认存放在磁盘上。当 Reducer 的输入文件已定时，整个 Shuffle 过程才最终结束。

7.3.4 性能差的主要原因

宏观上, Hadoop 的每个作业都要经历两个阶段: MapPhase 和 ReducePhase。对于 MapPhase, 又主要包含 4 个子阶段: 从磁盘上读数据→执行 Map 函数→Combine 结果→将结果写到本地磁盘上; 对于 ReducePhase, 同样包含 4 个子阶段: 从各个 MapTask 上读取相应的数据 (Shuffle) →Sort→执行 Reduce 函数→将结果写到 HDFS 中。

Hadoop 处理流程中的两个子阶段严重降低了其性能。一方面, Map 阶段产生的中间结果要写到磁盘上, 这样做的主要目的是提高系统的可靠性, 但代价是降低了系统性能; 另一方面, Shuffle 阶段采用 HTTP 协议从各个 MapTask 上远程复制结果, 这种设计思路同样降低了系统性能。

可以看出, 磁盘读/写速度慢是导致 MapReduce 性能差的主要原因。Spark 恰好看到了内存容量的增大和成本的降低, 决定用一个基于内存的架构去替代 MapReduce, 在性能上有了极大的提升。

7.4 Spark 架构和原理^①

7.4.1 Spark 的起源和特点

Spark 是发源于美国加利福尼亚大学伯克利分校 AMPLab 的集群计算平台。它立足于内存计算, 从多迭代批量处理出发, 兼收并蓄数据仓库、流处理和图计算等多种计算范式, 是罕见的“全能选手”。

1. 内存发展趋势

磁盘由于其物理特性限制, 导致速度提升非常困难, 远远跟不上 CPU 和内存的发展速度。近十几年来, 内存的发展一直遵循摩尔定律, 价格一直下降, 而容量一直增加。现在的主流服务器, 几百 GB 或者几 TB 的内存都很常见, 内存的发展使得内存数据库得以实现, 如著名的 VoltDB。Spark 也看好这种趋势, 所以设计的是一个基于内存的分布式处理软件, 也就是说 Spark 的目标是取代 MapReduce。

2. Spark 的愿景

当前开源社区针对不同的场景, 存在多种引擎, 如 Hadoop、Cassandra、Mesos 等。Spark 的愿景是做一个统一的引擎, 可以统一批处理、交互式处理、流处理等多种场景, 降低开发与运维难度, 如图 7.5 所示。

3. Spark 与 Hadoop 对比

(1) Spark 的中间数据存放在内存中, 对于迭代运算而言, 效率更高。

(2) Spark 更适合迭代运算比较多的数据挖掘和机器学习运算, 因为在 Spark 里有 RDD 的抽象概念。

(3) Spark 比 Hadoop 更通用。

(4) Spark 提供的数据集操作类型有很多, 而 Hadoop 只提供了 Map 和 Reduce 两种操作。

^① 参考 <http://tech.uc.cn/?p=2116>。

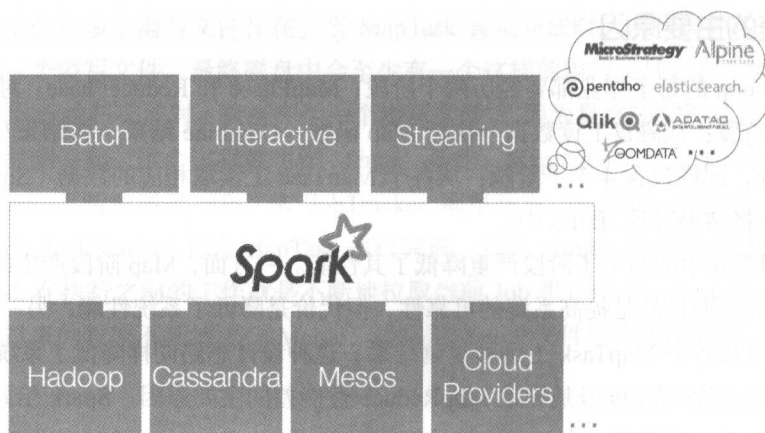


图 7.5

(5) 容错性。在分布式数据集计算时通过 Checkpoint 来实现容错。

(6) 可用性。Spark 通过提供丰富的 Scala、Java、Python API 及交互式 Shell 来提高可用性。

4. Spark 与 Hadoop 结合

Spark 可以直接对 HDFS 进行数据读/写，同样支持 Spark on YARN。Spark 可以和 MapReduce 运行在同一集群中，共享存储资源与计算。

5. Spark 的适用场景

Spark 是基于内存的迭代计算框架，适用于需要多次操作特定数据集的场合。需要反复操作的次数越多，需要读取的数据量越大，性能提升越大；数据量小但是计算密集度较大的场合，性能提升就相对较小。

由于 RDD 的特性，Spark 不适合那种异步细粒度更新状态的应用，如 Web 服务的存储或者增量的 Web 爬虫和索引。

总的来说，Spark 的适用范围较广，且较为通用。

7.4.2 Spark 的核心概念

1. 概念

1) 基本概念 (Basic Concepts)

- RDD: Resilient Distributed Dataset，弹性分布式数据集。
- Operation: 作用于 RDD 的各种操作，包括 Transformation 和 Action。
- Job: 作业，一个 Job 包含多个 RDD 及作用于相应 RDD 上的各种 Operation。
- Stage: 一个作业分为多个阶段。
- Partition: 数据分区，一个 RDD 中的数据可以分成多个不同的区。
- DAG: Directed Acycle Graph，有向无环图，反映 RDD 之间的依赖关系。

- **Narrow Dependency:** 窄依赖, 子 RDD 依赖于父 RDD 中固定的 Data Partition。
- **Wide Dependency:** 宽依赖, 子 RDD 对父 RDD 中的所有 Data Partition 都有依赖。
- **Caching Management:** 缓存管理, 对 RDD 的中间计算结果进行缓存管理, 以加快整体的处理速度。

2) 编程模型 (Programming Model)

RDD 是只读的数据分区集合, 注意是数据集。

作用于 RDD 上的 Operation 分为 Transformation 和 Action。经 Transformation 处理之后, 数据集内容会发生更改, 由数据集 A 转换成数据集 B; 而经 Action 处理之后, 数据集内容会被归约为一个具体的数值。

只有当 RDD 上有 Action 时, 该 RDD 及其父 RDD 上的所有 Operation 才会被提交到 Cluster 中真正被执行。

Spark 代码演示如图 7.6 所示。

```
val sc = new SparkContext("Spark://...", "MyJob", home, jars)
val file = sc.textFile("hdfs://...")
val errors = file.filter(_.contains("ERROR"))
errors.cache()
errors.count()
```

图 7.6

3) 运行态 (Runtime View)

不管是什么样的静态模型, 其在动态运行的时候无外乎由进程、线程组成。

用 Spark 的术语来说, Static View 称为 Dataset View, 而 Dynamic View 称为 Partition View, 二者之间的关系如图 7.7 所示。

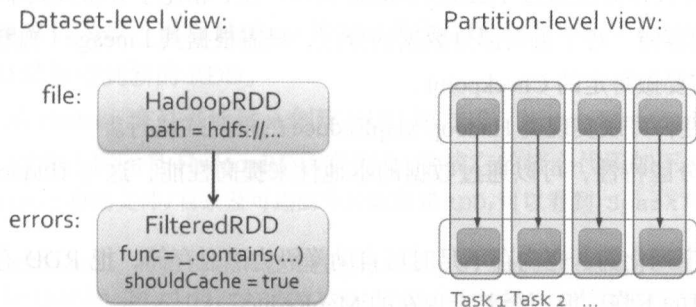


图 7.7

4) 部署 (Deployment View)

当有 Action 作用于某 RDD 时, 该 Action 会作为一个 Job 被提交。

在提交的过程中, DAGScheduler 模块介入运算, 计算 RDD 之间的依赖关系。RDD 之间的依赖关系就形成了 DAG。

每个 Job 被分为多个 Stage。划分 Stage 的一个主要依据是当前计算因子的输入是否是确定的，如果是则将其分在同一个 Stage，从而避免多个 Stage 之间的消息传递开销。

当 Stage 被提交之后，由 TaskScheduler 来根据 Stage 计算所需的 Task，并将 Task 提交到对应的 Worker。

Spark 支持 Standalone、Mesos、YARN 等部署模式，这些部署模式将作为 TaskScheduler 的初始化入参。

5) Resilient Distributed Dataset (RDD) 弹性分布数据集

RDD 是 Spark 的最基本抽象，是对分布式内存的抽象使用，以操作本地集合的方式来操作分布式数据集的抽象实现。RDD 是 Spark 最核心的内容，它表示已被分区、不可变的、能够被并行操作的数据集，不同的数据集格式对应不同的 RDD 实现。RDD 必须是可序列化的。RDD 可以缓存到内存中，每次对 RDD 数据集的操作结果都可以存放到内存中，下一个操作可以直接从内存中输入，省去了 MapReduce 大量的磁盘 I/O 操作。这对于迭代运算比较常见的机器学习算法、交互式数据挖掘来说，效率提升比较大。

① RDD 的特点

- 它是在集群节点上的不可变的、已分区的集合对象。
- 通过并行转换的方式来创建。
- 失败自动重建。
- 可以控制存储级别（内存、磁盘等）来进行重用。
- 必须是可序列化的。
- 是静态类型的。

② RDD 的优势

- RDD 只能从持久存储或通过 Transformation 操作产生，相比于分布式共享内存（DSM），可以更高效地实现容错。对于丢失部分数据的分区，只需根据其 Lineage（血统）就可以重新计算出来，而不需要做特定的 Checkpoint。
- RDD 的不变性，可以实现类 Hadoop MapReduce 的推测式执行。
- RDD 的数据分区特性，可以通过数据的本地性来提高性能，这与 Hadoop MapReduce 是一样的。
- RDD 都是可序列化的，在内存不足时可自动降级为磁盘存储，把 RDD 存储于磁盘上。这时性能会有明显的下降，但不会差于现在的 MapReduce。

③ RDD 的存储与分区

用户可以选择不同的存储级别存储 RDD 以便重用。

当前 RDD 默认存储于内存中，但当内存不足时，RDD 会溢出到磁盘上。

RDD 是根据每条记录的 Key 进行分区的（如 Hash 分区），具有相同 Key 的数据会存储在同一个节点上，以保证两个数据集在 Join 时能高效进行。

④ RDD 的内部表示

RDD 由以下几个主要部分组成。

- partitions: partition 集合, 一个 RDD 中有多个 data partition。
- dependencies: RDD 依赖关系。
- compute(partition): 对于给定的数据集, 需要进行哪些计算。
- preferredLocations: 对于 data partition 的位置偏好。
- partitioner: 对于计算出来的数据结果如何分发。

⑤ RDD 的存储级别

RDD 根据 useDisk、useMemory、deserialized、replication 4 个参数的组合提供了 11 种存储级别,

如下:

```
val NONE = new StorageLevel(false, false, false)
val DISK_ONLY = new StorageLevel(true, false, false)
val DISK_ONLY_2 = new StorageLevel(true, false, false, 2)
val MEMORY_ONLY = new StorageLevel(false, true, true)
val MEMORY_ONLY_2 = new StorageLevel(false, true, true, 2)
val MEMORY_ONLY_SER = new StorageLevel(false, true, false)
val MEMORY_ONLY_SER_2 = new StorageLevel(false, true, false, 2)
val MEMORY_AND_DISK = new StorageLevel(true, true, true)
val MEMORY_AND_DISK_2 = new StorageLevel(true, true, true, 2)
val MEMORY_AND_DISK_SER = new StorageLevel(true, true, false)
val MEMORY_AND_DISK_SER_2 = new StorageLevel(true, true, false, 2)
```

RDD 定义了各种操作, 不同类型的数据由不同的 RDD 类抽象表示, 不同的操作也由 RDD 进行抽象实现。

⑥ RDD 的创建

RDD 有两种创建方式:

- (1) 从 Hadoop 文件系统 (或与 Hadoop 兼容的其他存储系统) 输入 (如 HDFS) 创建。
- (2) 从父 RDD 转换得到新的 RDD。

下面来看一下从 Hadoop 文件系统输入创建 RDD 的方式, 如 `val file=spark.textFile("hdfs://...")`, 其中 file 变量就是 RDD (实际是 HadoopRDD 实例), 生成它的核心代码如下:

// SparkContext 根据文件/目录及可选的分片数创建 RDD, 可以看到, Spark 与 Hadoop MapReduce 很像

```
// 需要 InputFormat, 其实 Spark 使用的是 Hadoop 的 InputFormat、Writable 类型
def textFile(path: String, minSplits: Int = defaultMinSplits): RDD[String] =
{
    hadoopFile(path, classOf[TextInputFormat], classOf[LongWritable],
        classOf[Text], minSplits) .map(pair => pair._2.toString) }
// 根据 Hadoop 配置及 InputFormat 等创建 HadoopRDD
new HadoopRDD(this, conf, inputFormatClass, keyClass, valueClass, minSplits)
```

在对 RDD 进行计算时, RDD 从 HDFS 读取数据时与 Hadoop MapReduce 几乎是一样的, 如下:

// 根据 Hadoop 配置和分片从 InputFormat 中获取 RecordReader 进行数据读取

```

reader = fmt.getRecordReader(split.inputSplit.value, conf, Reporter.NULL)
val key: K = reader.createKey()
val value: V = reader.createValue()
//使用 Hadoop MapReduce 的 RecordReader 读取数据，每个 Key/Value 对以元组形式返回
override def getNext() = {
  try {
    finished = !reader.next(key, value)
  } catch {
    case eof: EOFException =>
      finished = true
  }
  (key, value)
}

```

⑦ RDD 的转换与操作

对于 RDD 有两种计算方式：转换（Transformations，返回值还是一个 RDD）与操作（Actions，返回值不是一个 RDD）。

下面通过一个例子来说明 Transformations 与 Actions 在 Spark 中的使用，如图 7.8 所示。代码如下：

```

val sc = new SparkContext(master, "Example", System.getenv("SPARK_HOME"),
  Seq(System.getenv("SPARK_TEST_JAR")))
val rdd_A = sc.textFile(hdfs://.....)
val rdd_B = rdd_A.flatMap((line => line.split("\\s+"))).map(word => (word, 1))
val rdd_C = sc.textFile(hdfs://.....)
val rdd_D = rdd_C.map(line => (line.substring(10), 1))
val rdd_E = rdd_D.reduceByKey((a, b) => a + b)
val rdd_F = rdd_B.join(rdd_E)
rdd_F.saveAsSequenceFile(hdfs://.....)

```

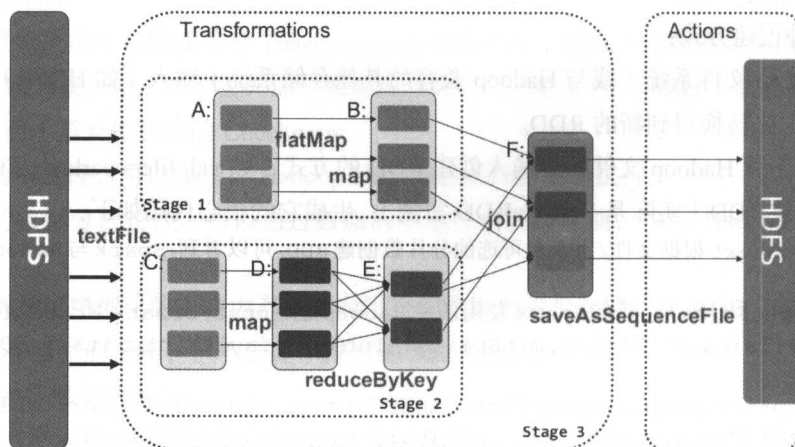


图 7.8

⑧ Lineage（血统）

利用内存加快数据加载，在众多其他的 In-Memory 类数据库或 Cache 类系统中也有实现，而

Spark 的特殊之处在于它处理分布式运算环境下的数据容错性（节点失效、数据丢失）问题时采用的方案。为了保证 RDD 中数据的鲁棒性，RDD 数据集通过所谓的血统关系（Lineage）记住了它是如何从其他 RDD 中演变过来的。相比其他系统的细颗粒度的内存数据更新级别的备份或者 Log 机制，RDD 的 Lineage 记录的是粗颗粒度的特定数据转换（Transformations）操作。当这个 RDD 的部分分区数据丢失时，它可以通过 Lineage 获取足够的信息来重新运算和恢复丢失的数据分区。这种粗颗粒度的数据模型限制了 Spark 的适用场合，但相比细颗粒度的数据模型，也带来了性能上的提升。

RDD 在 Lineage 依赖方面分为窄依赖与宽依赖两种，用来解决数据容错的高效性。窄依赖是指父 RDD 的每个分区最多被一个子 RDD 的分区所用，表现为一个父 RDD 的分区对应于一个子 RDD 的分区或多个父 RDD 的分区对应于一个子 RDD 的分区，也就是说一个父 RDD 的一个分区不可能对应一个子 RDD 的多个分区。宽依赖是指子 RDD 的分区依赖于父 RDD 的多个分区或所有分区，也就是说，存在一个父 RDD 的一个分区对应一个子 RDD 的多个分区。

⑨ 容错性

在 RDD 计算中，通过 Checkpoint 进行容错有两种方式：一种是 Checkpoint data，另一种是 Logging the updates。用户可以控制采用哪种方式来实现容错，默认采用 Logging the updates 方式，通过记录跟踪所有生成 RDD 的转换（Transformations），也就是记录每个 RDD 的 Lineage（血统），来重新计算生成丢失的分区数据。

6) 缓存机制（Caching）

RDD 的中间计算结果可以被缓存起来。缓存优先选择内存，如果内存不足，则会被写入磁盘中。根据 LRU（Last-Recent Update）来决定哪些内容继续保存在内存，哪些内容保存到磁盘。

7) 集群管理和资源管理

Task 运行在 Cluster 之上，除了 Spark 自身提供的 Standalone 部署模式外，还内在支持 YARN 和 Mesos。

YARN 负责计算资源的调度和监控。YARN 会自动重启失效的 Task，如果有新的 Node 加入，则会自动重分布 Task。

Spark on YARN 在 Spark 0.6 版本时引入，但真正可用的是 branch-0.8 版本。Spark on YARN 遵循 YARN 的官方规范实现，得益于 Spark 天生支持多种 Scheduler 和 Executor 的良好设计，对 YARN 的支持也就非常容易。Spark on YARN 的大致框图如图 7.9 所示。

让 Spark 运行于 YARN 之上与 Hadoop 共用集群资源，可以提高资源利用率。

2. Spark 机制详解

1) 编程接口

Spark 通过与编程语言集成的方式暴露 RDD 的操作，类似于 DryadLINQ 和 FlumeJava，每个数据集都表示为 RDD 对象，对数据集的操作就表示为对 RDD 对象的操作。Spark 主要的编程语言是 Scala，选择 Scala 是因为它的简洁性（Scala 可以很方便地在交互式下使用）和性能（JVM 上的静态强类型语言）。

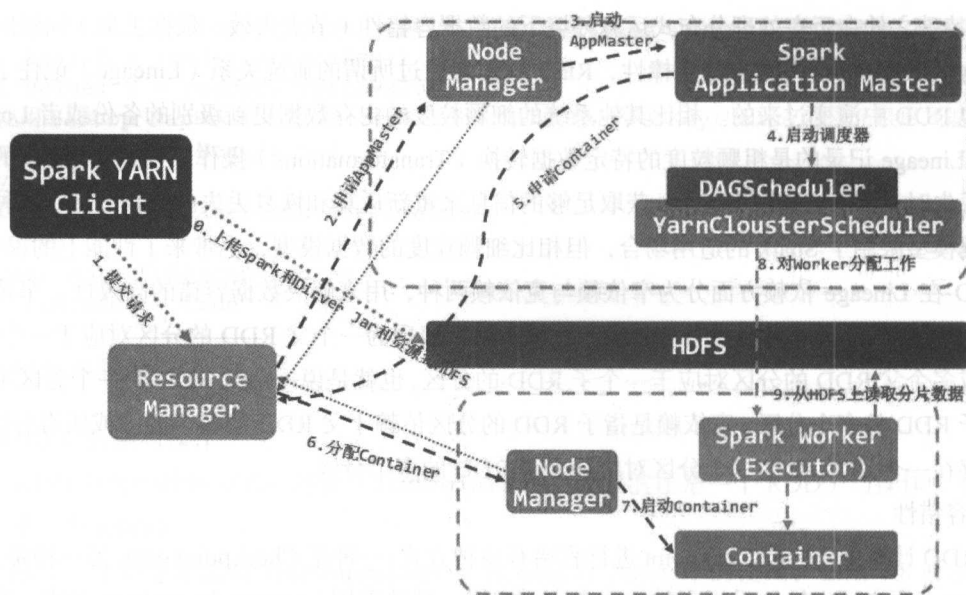


图 7.9

Spark 和 Hadoop MapReduce 类似，由 Master（类似于 MapReduce 的 JobTracker）和 Worker（Spark 的 Slave 工作节点）组成。用户编写的 Spark 程序被称为 Driver 程序，Driver 程序会连接 Master 并定义对各 RDD 的转换与操作，而对 RDD 的转换与操作通过 Scala 闭包（字面量函数）来表示，Scala 使用 Java 对象来表示闭包且都是可序列化的，以此把对 RDD 的闭包操作发送到各 Worker 节点。Worker 存储数据分块和享有集群内存，是运行在工作节点上的守护进程，当它收到对 RDD 的操作时，根据数据分片信息进行本地化数据操作，生成新的数据分片、返回结果或把 RDD 写入存储系统，如图 7.10 所示。

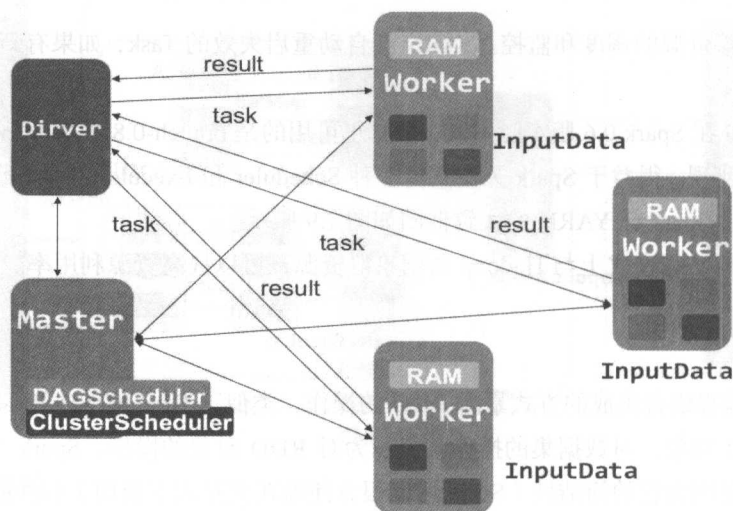


图 7.10

① Scala

Spark 使用 Scala 开发,默认使用 Scala 作为编程语言。编写 Spark 程序比编写 Hadoop MapReduce 程序要简单得多。Spark 提供了 Spark-Shell,可以在 Spark-Shell 中测试程序。编写 Spark 程序的一般步骤就是创建或使用 (SparkContext) 实例,即使用 SparkContext 创建 RDD,然后对 RDD 进行操作。如下:

```
val sc = new SparkContext(master,appName,[sparkHome],[jars])
val textFile = sc.textFile("hdfs://.....")
textFile.map(...).filter(.....).....
```

② Java

Spark 支持 Java 编程,但没有了 Spark-Shell 这样方便的工具,其他与 Scala 编程是一样的。因为都是 JVM 上的语言,所以 Scala 与 Java 可以互操作,Java 编程接口其实就是对 Scala 的封装。如下:

```
JavaSparkContext sc = new JavaSparkContext(...);
JavaRDD lines = ctx.textFile("hdfs://...");
JavaRDD words = lines.flatMap(
    new FlatMapFunction<String,String>() {
        public Iterable call(String s) {
            return Arrays.asList(s.split(" "));
        }
    }
);
```

③ Python

现在 Spark 也提供了 Python 编程接口,Spark 使用 Py4J 来实现 Python 与 Java 的互操作,从而实现使用 Python 编写 Spark 程序。Spark 同样提供了 PySpark,它是一个 Spark 的 Python-Shell,可以以交互的方式使用 Python 编写 Spark 程序。如下:

```
from pyspark import SparkContext
sc = SparkContext("local","Job Name",pyFiles=['MyFile.py','lib.zip'],'app.egg'])
words = sc.textFile("/usr/share/dict/words")
words.filter(lambda w: w.startswith("spar")).take(5)
```

④ Spark SQL^①

Spark 之所以长盛不衰,Spark SQL 起到的作用非常大。经过数据库多年的使用和发展,以及 SQL 的简单易用,所以一个处理平台具备 SQL 能力是基本要求。

Spark 诞生之后,立刻就有 Hive on Spark/Shark 等项目想做 SQL on Spark。Hive on Spark/Shark 引擎由于在机制上是架构在 Spark 引擎外面的,所以优化能力非常有限,性能不理想。因此,DataBricks 最初只是想替换掉 Shark,提供一个更高效的 SQL on Spark。在这一过程中推出两个 API,分别是 DataSource API 和 DataFrame API,将 Spark 的易用性及生态对 Spark 的支持提升到一个新的高度。

① 参考 <http://www.csdn.net/article/2015-06-15/2824958>。

⑤ DataSource API

Spark 1.2（2014 年 12 月）在发布 DataSource API 短短 4 个月后，就已经获得 10 余家厂商的支持。DataSource API 的主要特点如下：

- API 适配简单易用；支持针对数据源增加优化规则，实现最优数据访问。
- API 是事实标准，是社区接纳的标准。

DataSource API 只需实现 3 个接口，只需 500 行代码即可实现与 Apache Avro 文件对接。这三个接口介绍如下。

（1）RelationProvider。这个 API 提供给外部数据源来负责实现，接受 Parse 后传入的参数，生成对应的 External Relation，其本质上是一个反射生产外部数据源 Relation 的接口。

```
trait RelationProvider {
  /**
   * Returns a new base relation with the given parameters.
   * Note: the parameters' keywords are case insensitive and this insensitivity
   is enforced
   * by the Map that is passed to the function.
   */
  def createRelation(sqlContext: SQLContext, parameters: Map[String, String]):
  BaseRelation
}
```

（2）BaseRelation。这是外部数据源的抽象，里面存放了 Schema 的映射，以及扫描数据的规则。

```
abstract class BaseRelation {
  def sqlContext: SQLContext
  def schema: StructType
  abstract class PrunedFilteredScan extends BaseRelation {
    def buildScan(requiredColumns: Array[String], filters: Array[Filter]): RDD
[Row]
  }
}
```

（3）BuildScan。默认支持 4 种 BaseRelation，分别为 TableScan、PrunedScan、PrunedFilterScan 和 CatalystScan。

a. TableScan：默认的 Scan 策略。

b. PrunedScan：这里可以传入指定的列，不需要的列不会从外部数据源加载，从而节省 I/O，提高效率。

c. PrunedFilterScan：在列裁剪的基础上加入 Filter（过滤）机制，在加载数据的同时进行过滤，而不是在客户端请求返回时进行过滤。

d. CatalystScan：支持根据优化器优化的表达式来执行，通过 Data Source API 吸引了大量第三方场景对接，形成了一个非常大的 Spark 生态。

⑥ DataFrame API

下面来看一段分别使用三种接口——MapReduce、RDD 和 DataFrame API 实现同一个功能的程序。首先是 MapReduce 接口：

```

private IntWritable one =
    new IntWritable(1)
private IntWritable output =
    new IntWritable()
protected void map(
    LongWritable key,
    Text value,
    Context context) {
    String[] fields = value.split("\t")
    output.set(Integer.parseInt(fields[1]))
    context.write(one, output)
}

IntWritable one = new IntWritable(1)
DoubleWritable average = new DoubleWritable()

protected void reduce(
    IntWritable key,
    Iterable<IntWritable> values,
    Context context) {
    int sum = 0
    int count = 0
    for(IntWritable value : values) {
        sum += value.get()
        count++
    }
    average.set(sum / (double) count)
    context.write(key, average)
}

```

然后是 RDD 接口：

```

data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [int(x[1]), 1])) \
    .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
    .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
    .collect()

```

最后是 DataFrame API 接口：

```

sqlCtx.table("people") \
    .groupBy("name") \
    .agg("name", avg("age")) \
    .map(lambda ...) \
    .collect()

```

通过三段代码对比，可以发现，DataFrame API 大大简化了编程的难度，让不熟悉 Scala 的人也能使用。

⑦ DataFrame

与 RDD 类似，DataFrame 也是一个分布式数据容器。然而 DataFrame 更像传统数据库的二维表，除了数据外，还掌握着数据的结构信息，即 Schema。同时，与 Hive 类似，DataFrame 也支持嵌套数据类型（struct、array 和 map）。从 API 易用性的角度来看，DataFrame API 提供的是一套更高抽象的关系操作接口，比函数式的 RDD API 更加友好、门槛更低。由于与 R 和 Pandas 的 DataFrame 类似，因而 Spark DataFrame 很好地继承了传统单机数据分析的开发体验。

图 7.11 直观地体现了 DataFrame 和 RDD 的区别。左侧的 RDD[Person]虽然以 Person 为类型参数，但 Spark 框架本身并不了解 Person 类的内部结构。而右侧的 DataFrame 却提供了详细的结构信息，使得 Spark SQL 可以清楚地知道该数据集中包含哪些列、每列的名称和类型各是什么。了解了这些信息，Spark SQL 的查询优化器就可以进行有针对性的优化。

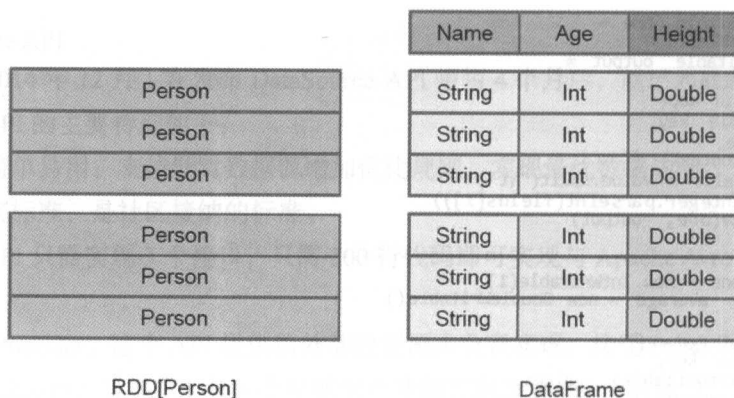


图 7.11

③ 查询优化

Spark SQL 的第三个目标就是让查询优化器帮助优化执行效率，解放开发者的生产力，让新手也能写出高效的程序。

DataFrame 的背后是 Spark SQL 的全套查询优化引擎，其整体架构如图 7.12 所示。通过 SQL/HiveQL Parser 或 DataFrame API 构造的逻辑执行计划经分析后再经优化得到优化执行计划，接着再转为物理执行计划，最终转换为 RDD DAG 在 Spark 引擎上执行。

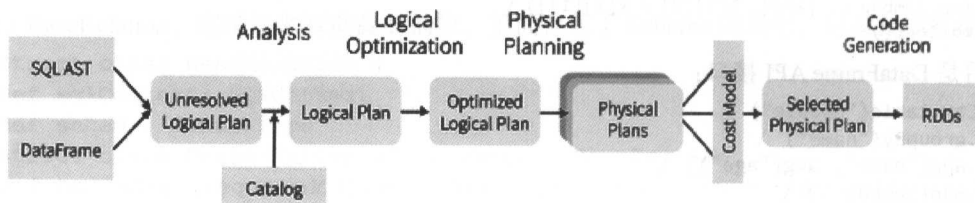


图 7.12

为了说明查询优化，我们来看图 7.13 展示的人口数据分析的示例。图中构造了两个 DataFrame，将它们 Join 之后又执行了一次 Filter 操作。如果原封不动地执行这个计划，那么最终的执行效率是不高的。因为 Join 是一个代价较大的操作，也可能会产生一个较大的数据集。如果能将 Filter 下推到 Join 下方，先对 DataFrame 进行过滤，再 Join 过滤后的较小的结果集，则可以有效地缩短执行时间。而 Spark SQL 的查询优化器正是这样做的。简言之，逻辑查询计划优化就是一个利用基于关系代数的等价变换，将高成本的操作替换为低成本的操作的过程。

得到的优化执行计划在转换成物理执行计划的过程中，还可以根据具体数据源的特性将过滤条件下推至数据源内。最右侧的物理执行计划中的 Filter 之所以消失不见，就是因为融入了用于执行最终的读取操作的表扫描节点内。

对于普通开发者而言，查询优化器的意义在于，即便经验并不丰富的程序员写出的次优的查询，也可以被尽可能地转换为高效的形式予以执行。

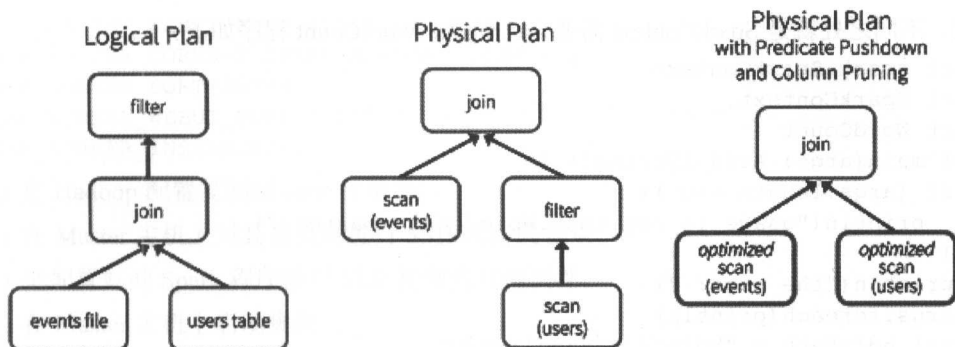


图 7.13

在 Spark 2.0 中统一用一套 API 同时处理批处理场景、交互式查询场景、机器学习场景及流处理场景。DataFrame 的结构会发生变化，形象地形容是由固定的会变成无穷大小的，参考图 7.14。

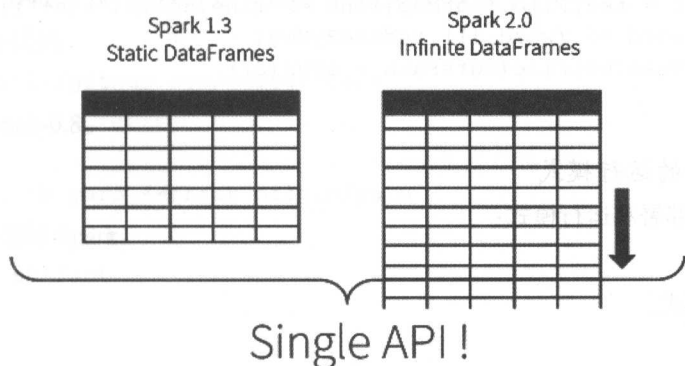


图 7.14

2) 使用 Spark-Shell

Spark-Shell 的使用很简单，当 Spark 以 Standalone 模式运行后，使用 \$SPARK_HOME/Spark-Shell 进入 Shell 即可。在 Spark-Shell 中，SparkContext 已经创建好，实例名为 sc，可以直接使用。另外，需要注意的是，在 Standalone 模式下，Spark 默认使用的调度器是 FIFO 调度器，而 Spark-Shell 作为一个 Spark 程序一直运行在 Spark 上，其他 Spark 程序就只能排队等待，也就是说同一时间只能有一个 Spark-Shell 在运行。在 Spark-Shell 上写程序非常简单，就像在 Scala-Shell 上写程序一样。示例代码如下：

```
scala> val textFile = sc.textFile("hdfs://hadoop1:2323/user/data")
textFile: spark.RDD[String] = spark.MappedRDD@2ee9b6e3
scala> textFile.count() // Number of items in this RDD
res0: Long = 21374
scala> textFile.first() // First item in this RDD
res1: String = # Spark
```

3) 编写 Driver 程序

在 Spark 中，Spark 程序被称为 Driver 程序。编写 Driver 程序很简单，几乎与在 Spark-Shell 上写

程序相同，不同之处就是 `SparkContext` 需要自己创建。`WordCount` 程序如下：

```
import spark.SparkContext
import SparkContext._
object WordCount {
  def main(args: Array[String]) {
    if (args.length == 0) {
      println("usage is org.test.WordCount <master>")
    }
    println("the args: ")
    args.foreach(println)
    val hdfsPath = "hdfs://hadoop1:8020"
    // create the SparkContext, args(0) 由 YARN 传入 appMaster 地址
    val sc = new SparkContext(args(0), "WordCount",
      System.getenv("SPARK_HOME"), Seq(System.getenv("SPARK_TEST_JAR")))
    val textFile = sc.textFile(hdfsPath + args(1))
    val result = textFile.flatMap(line => line.split("\\s+"))
      .map(word => (word, 1)).reduceByKey(_ + _)
    result.saveAsTextFile(hdfsPath + args(2))
  }
}
```

4) Spark 支持的运行模式

Spark 支持多种部署和运行模式：

- 本地模式。
- Standalone 模式。
- Mesos 模式。
- YARN 模式。

① Standalone 模式

为了方便推广使用，Spark 提供了 Standalone 模式。Spark 一开始就设计运行于 Apache Mesos 资源管理框架上，这是非常好的设计，但是却带来了部署测试的复杂性。为了让 Spark 更方便地部署和尝试，Spark 提供了 Standalone 运行模式，它由一个 Spark Master 和多个 Spark Worker 组成，与 Hadoop MapReduce 1 类似，就连集群启动方式也几乎是一样的。

以 Standalone 模式运行 Spark 集群，步骤如下。

(1) 下载 Scala 2.9.3，并配置 `SCALA_HOME`。

(2) 下载 Spark 代码（可以使用源码编译，也可以下载编译好的版本，这里下载编译好的版本），下载地址为 <http://spark-project.org/download/spark-0.7.3-prebuilt-cdh4.tgz>。

(3) 解压 `spark-0.7.3-prebuilt-cdh4.tgz` 安装包。

(4) 修改环境变量（修改 `conf/` 配置工作节点的主机名/`spark-env.sh`）。

```
SCALA_HOME=/home/spark/scala-2.9.3
JAVA_HOME=/home/spark/jdk1.6.0_45
SPARK_MASTER_IP=spark1
SPARK_MASTER_PORT=30111
```

```
SPARK_MASTER_WEBUI_PORT=30118
SPARK_WORKER_CORES=2 SPARK_WORKER_MEMORY=4g
SPARK_WORKER_PORT=30333
SPARK_WORKER_WEBUI_PORT=30119
SPARK_WORKER_INSTANCES=1
```

- (5) 把 Hadoop 配置复制到 conf 目录下。
- (6) 在 Master 主机上对其他机器进行 SSH 无密码登录。
- (7) 把配置好的 Spark 程序使用 SCP 复制到其他机器。
- (8) 在 Master 主机上启动集群。

```
$SPARK_HOME/start-all.sh
```

② YARN 模式

Spark-Shell 现在还不支持 YARN 模式, 要使用 YARN 模式运行, 需要把 Spark 程序全部打包成一个 JAR 包提交到 YARN 上。YARN 模式运行 Spark 的步骤如下。

- (1) 下载 Spark 代码。

```
git clone git://github.com/mesos/spark
```

- (2) 切换到 branch-0.8。

```
cd spark
git checkout -b yarn --track origin/yarn
```

- (3) 使用 SBT 编译 Spark。

```
$SPARK_HOME/sbt/sbt
> package
> assembly
```

- (4) 把 Hadoop YARN 配置复制到 conf 目录下。

- (5) 运行测试。

```
SPARK_JAR=./core/target/scala-2.9.3/spark-core-assembly-0.8.0-SNAPSHOT.jar \
./run spark.deploy.yarn.Client --jar examples/target/scala-2.9.3/ \
--class spark.examples.SparkPi --args yarn-standalone
```

5) Job 运行机制

下面以 WordCount 为例, 详细说明 Spark 创建和运行 Job 的过程, 重点是进程和线程的创建。

① 实验环境搭建

在进行后续操作前, 请确保下列条件已满足:

- (1) 下载 Spark Binary 0.9.1。
- (2) 安装 Scala。
- (3) 安装 SBT。
- (4) 安装 Java。

② 启动 Spark-Shell

③ 单机模式运行, 即 Local 模式

Local 模式运行非常简单，只需执行以下命令即可（假设当前目录是\$SPARK_HOME）：

```
MASTER=local bin/spark-shell
```

其中，“MASTER=local”表明当前运行在单机模式下。

④ Local Cluster 模式运行

Local Cluster 模式是一种伪 Cluster 模式，在单机环境下模拟 Standalone 的集群，启动顺序如下。

（1）启动 Master。

```
$SPARK_HOME/sbin/start-master.sh
```

注意运行时的输出，日志默认保存在\$SPARK_HOME/logs 目录下。

Master 主要运行 org.apache.spark.deploy.master.Master 类，在 8080 端口启动监听，监听日志如下：

```
14/04/21 10:14:15 INFO Slf4jLogger: Slf4jLogger started
14/04/21 10:14:15 INFO Remoting: Starting remoting
14/04/21 10:14:15 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkMaster@localhost:7077]
14/04/21 10:14:15 INFO Master: Starting Spark master at spark://localhost:7077
14/04/21 10:14:15 INFO MasterWebUI: Started Master web UI at http://localhost:8080
14/04/21 10:14:15 INFO Master: I have been elected leader! New state: ALIVE
```

修改配置：

- 进入\$SPARK_HOME/conf 目录。
- 将 spark-env.sh.template 重命名为 spark-env.sh。
- 修改 spark-env.sh，添加如下内容。

```
export SPARK_MASTER_IP=localhost
export SPARK_LOCAL_IP=localhost
```

（2）运行 Worker。

```
bin/spark-class org.apache.spark.deploy.worker.Worker
spark://localhost:7077 -i 127.0.0.1 -c 1 -m 512M
```

Worker 启动完成，连接到 Master。打开 Maser 的 Web UI，即可看到连接的 Worker。Master Web UI 的监听地址是 http://localhost:8080。

（3）启动 Spark-Shell。

```
MASTER=spark://localhost:7077 bin/spark-shell
```

如果一切顺利，将看到如下提示信息：

```
Created spark context..
Spark context available as sc.
```

可以用浏览器打开 localhost:4040 来查看如下内容：Stages、Storage、Environment 和 Executors。

⑤ WordCount

上述环境准备好后，在 Spark-Shell 中运行一下最简单的例子。在 Spark-Shell 中输入如下代码：

```
scala>sc.textFile("README.md").filter(_.contains("Spark")).count
```

上述代码统计在 README.md 中含有的 Spark 行数。

⑥ 部署过程详解

Spark 布置环境中的组件构成如图 7.15 所示。

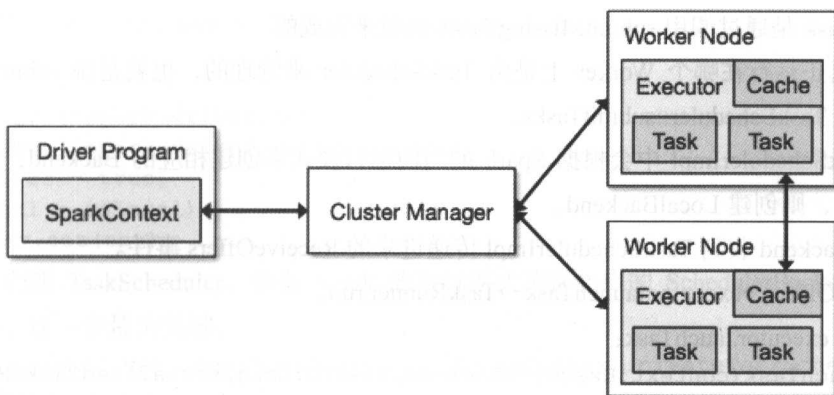


图 7.15

各组件的含义如下。

- Driver Program: 简单来说, 其与在 Spark-Shell 中输入的 WordCount 语句相对应。
- Cluster Manager: 与 Master 相对应, 主要起到部署和管理的作用。
- Worker Node: 与 Master 相比, 这是 Slave 节点。其上运行各个 Executor, Executor 可以与线程相对应。Executor 处理两种基本的业务逻辑: 一种是 Driver Program; 另一种是 Job 在提交之后拆分成各个 Stage, 每个 Stage 可以运行一个或多个 Task。

其中, 在集群 (Cluster) 方式下, Cluster Manager 运行在一个 JVM 进程中, 而 Worker 运行在另一个 JVM 进程中。在 Local Cluster 中, 这些 JVM 进程都运行在同一台机器中。如果是集群模式 (Standalone、Mesos、YARN 集群), Worker 与 Master 或分布于不同的主机之上。

⑦ Job 的生成和运行

Job 生成的简单流程如下:

- 首先, 应用程序创建 SparkContext 的实例, 如实例为 sc。
- 利用 SparkContext 的实例来创建 RDD。
- 经过一连串 Transformation 操作, 原始的 RDD 转换为其他类型的 RDD。
- 当 Action 作用于转换之后的 RDD 时, 会调用 SparkContext 的 runJob 方法。
- runJob 方法的调用是后面一连串反应的起点, 关键性的跃变就发生在此处。

调用路径大致如下:

- sc.runJob→dagScheduler.runJob→submitJob。
- DAGScheduler::submitJob 会创建 JobSubmitted 的 event 发送给内嵌类 eventProcessActor。
- eventProcessActor 在接收到 JobSubmitted 后, 调用 processEvent 处理函数。
- Job 到 Stage 的转换, 生成 finalStage 并提交运行, 关键是调用 submitStage。
- 在 submitStage 中会计算 Stage 之间的依赖关系, 分为宽依赖和窄依赖。
- 如果在计算中发现当前的 Stage 没有任何依赖, 或者所有依赖都已经准备完毕, 则提交 Task。

- 提交 Task 是通过调用 submitMissingTasks 函数来完成的。
- Task 真正运行在哪个 Worker 上是由 TaskScheduler 来管理的，也就是说 submitMissingTasks 会调用 TaskScheduler::submitTasks。
- 在 TaskSchedulerImpl 中会根据 Spark 的当前运行模式来创建相应的 Backend，如果是在单机上运行，则创建 LocalBackend。
- LocalBackend 收到 TaskSchedulerImpl 传递进来的 ReceiveOffers 事件。
- receiveOffers→executor.launchTask→TaskRunner.run。

代码片段 executor.launchTask:

```
def launchTask(context:ExecutorBackend,taskId:Long,serializedTask:ByteBuffer){
    val tr = new TaskRunner(context,taskId,serializedTask)
    runningTasks.put(taskId,tr)
    threadPool.execute(tr)
}
```

最终的逻辑处理发生在 TaskRunner 这个 Executor 之内。将运算结果包装成 MapStatus，然后通过一系列的内部消息传递，反馈到 DAGScheduler。

6) Task 流程分析

接下来阐述在 TaskRunner 中执行的 Task 的业务逻辑是如何被调用到的。另外，试图讲清楚运行着的 Task 的输入数据从哪里获取、处理的结果返回哪里，以及如何返回。

① 准备

- Spark 已经安装完毕。
- Spark 运行在 Local 或 Local Cluster 模式。

② Driver Program 的初始化分析

初始化过程涉及的主要源文件如下。

- SparkContext.scala: 整个初始化过程的入口。
- SparkEnv.scala: 创建 BlockManager、MapOutputTrackerMaster、ConnectionManager 和 CacheManager。
- DAGScheduler.scala: 任务提交的入口，即将 Job 划分成各个 Stage 的关键。
- TaskSchedulerImpl.scala: 决定每个 Stage 可以运行几个 Task，每个 Task 分别在哪个 Executor 上运行。
- SchedulerBackend。
- 如果是最简单的单机运行模式，则看 LocalBackend.scala；如果是集群模式，则看源文件 SparkDeploySchedulerBackend。

初始化过程步骤详解如下。

步骤 1: 先根据初始化入参生成 SparkConf，再根据 SparkConf 创建 SparkEnv。SparkEnv 中主要包含 4 个关键性组件：BlockManager、MapOutputTracker、ShuffleFetcher 和 ConnectionManager。


```
private[spark] val env = SparkEnv.create(
  conf,
  "",
  conf.get("spark.driver.host"),
  conf.get("spark.driver.port").toInt,
  isDriver = true,
  isLocal = isLocal)
SparkEnv.set(env)
```

步骤 2: 创建 TaskScheduler, 根据 Spark 的运行模式选择相应的 SchedulerBackend, 同时启动 TaskScheduler。这一步最为关键。

```
private[spark] var taskScheduler = SparkContext.createTaskScheduler(this,
master, appName)
taskScheduler.start()
```

TaskScheduler.start 的目的是启动相应的 SchedulerBackend, 并启动定时器进行检测。

```
override def start() {
  backend.start()
  if (!isLocal && conf.getBoolean("spark.speculation", false)) {
    logInfo("Starting speculative execution thread")
    import sc.env.actorSystem.dispatcher
    sc.env.actorSystem.scheduler.schedule(SPECULATION_INTERVAL milliseconds,
      SPECULATION_INTERVAL milliseconds) {
      checkSpeculatableTasks()
    }
  }
}
```

步骤 3: 以步骤 2 中创建的 TaskScheduler 实例为入参创建 DAGScheduler, 并启动运行。

```
@volatile private[spark] var dagScheduler = new DAGScheduler(taskScheduler)
dagScheduler.start()
```

步骤 4: 启动 Web UI。

```
ui.start()
```

③ RDD 的转换过程

还是以最简单的 WordCount 为例说明 RDD 的转换过程。

```
sc.textFile("README.md").flatMap(line=>line.split(" ")).map(word => (word,1)).
reduceByKey(_ + _)
```

这一行简短的代码其实发生了很复杂的 RDD 转换, 下面详细解释每一步的转换过程和转换结果。

步骤 1: val rawFile = sc.textFile("README.md")。

textFile 先生成 HadoopRDD, 然后再通过 Map 操作生成 MappedRDD。如果在 Spark-Shell 中执行上述语句, 那么得到的结果可以证明刚才所做的分析。

```
scala> sc.textFile("README.md")
14/04/23 13:11:48 WARN SizeEstimator: Failed to check whether UseCompressedOops
is set; assuming yes
```

```
14/04/23 13:11:48 INFO MemoryStore: ensureFreeSpace(119741) called with
curMem=0,maxMem=311387750
```

```
14/04/23 13:11:48 INFO MemoryStore: Block broadcast_0 stored as values to memory
(estimated size 116.9 KB,free 296.8 MB)
```

```
14/04/23 13:11:48 DEBUG BlockManager: Put block broadcast_0 locally took 277 ms
```

```
14/04/23 13:11:48 DEBUG BlockManager: Put for block broadcast_0 without
replication took 281 ms
```

```
res0: org.apache.spark.rdd.RDD[String] = MappedRDD[1] at textFile at :13
```

步骤 2: `val splittedText = rawFile.flatMap(line => line.split(" "))`。

`flatMap` 将原来的 `MappedRDD` 转换成 `FlatMappedRDD`。

```
def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U] =
new FlatMappedRDD(this,sc.clean(f))
```

步骤 3: `val wordCount = splittedText.map(word => (word,1))`。

利用 `word` 生成相应的键值对，将步骤 2 得到的 `FlatMappedRDD` 转换成 `MappedRDD`。

步骤 4: `val reduceJob = wordCount.reduceByKey(_+_)`。这一步最为复杂。

步骤 2 和步骤 3 中用到的 `Operation` 全部定义在 `RDD.scala` 中，而这里用到的 `reduceByKey` 却在 `RDD.scala` 中见不到踪迹。`reduceByKey` 的定义出现在源文件 `PairRDDFunctions.scala` 里。

细心的读者一定会问：`reduceByKey` 是 `MappedRDD` 的属性和方法，怎么能被 `MappedRDD` 调用呢？其实这背后发生了一个隐式的转换，即将 `MappedRDD` 转换成 `PairRDDFunctions`。

```
implicit def rddToPairRDDFunctions[K: ClassTag,V: ClassTag](rdd: RDD[(K,V)]) =
new PairRDDFunctions(rdd)
```

再来看看 `reduceByKey` 的定义。

```
def reduceByKey(func: (V,V) => V): RDD[(K,V)] = {
  reduceByKey(defaultPartitioner(self),func)
}
def reduceByKey(partitioner: Partitioner,func: (V,V) => V): RDD[(K,V)] = {
  combineByKey[V]((v: V) => v,func,func,partitioner)
}
def combineByKey[C](createCombiner: V => C,
  mergeValue: (C,V) => C,
  mergeCombiners: (C,C) => C,
  partitioner: Partitioner,
  mapSideCombine: Boolean = true,
  serializerClass: String = null): RDD[(K,C)] = {
  if (getKeyClass().isArray) {
    if (mapSideCombine) {
      throw new SparkException("Cannot use map-side combining with array keys.")
    }
    if (partitioner.isInstanceOf[HashPartitioner]) {
      throw new SparkException("Default partitioner cannot partition array keys.")
    }
  }
  val aggregator = new Aggregator[K,V,C](createCombiner,mergeValue,
```

```

mergeCombiners)
  if (self.partitioner == Some(partitioner)) {
    self.mapPartitionsWithContext((context, iter) => {
      new InterruptibleIterator(context, aggregator.combineValuesByKey
(iter, context))
    }, preservesPartitioning = true)
  } else if (mapSideCombine) {
    val combined = self.mapPartitionsWithContext((context, iter) => {
      aggregator.combineValuesByKey(iter, context)
    }, preservesPartitioning = true)
    val partitioned = new ShuffledRDD[K, C, (K, C)](combined, partitioner)
    .setSerializer(serializerClass)
    partitioned.mapPartitionsWithContext((context, iter) => {
      new
InterruptibleIterator(context, aggregator.combineCombinersByKey(iter, context))
    }, preservesPartitioning = true)
  } else {
    // Don't apply map-side combiner.
    val values = new ShuffledRDD[K, V, (K, V)](self, partitioner).setSerializer
(serializerClass)
    values.mapPartitionsWithContext((context, iter) => {
      new
InterruptibleIterator(context, aggregator.combineValuesByKey(iter, context))
    }, preservesPartitioning = true)
  }
}

```

reduceByKey 最终会调用 combineByKey，在这个函数中，PairedRDDFunctions 会被转换成 ShuffleRDD，在调用 mapPartitionsWithContext 之后，shuffleRDD 被转换成 MapPartitionsRDD。日志输出能证明我们的分析，如下：

```

res1: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[8] at reduceByKey
at :13

```

④ RDD 转换小结

整个 RDD 转换过程为：HadoopRDD → MappedRDD → FlatMappedRDD → MappedRDD → PairRDDFunctions → ShuffleRDD → MapPartitionsRDD。

整个转换过程发生在任务提交之前。

⑤ 数据集操作分类

在对任务运行过程中的函数调用关系进行分析之前，先来探讨一个偏理论的问题：作用于 RDD 之上的转换可以抽象成哪几类？

对这个问题的解答将涉及数学知识。从理论抽象的角度来说，任务处理都可以归结为“input → processing → output”。input 和 output 对应于数据集 dataset。

在此基础上作一下简单的分类。

- one-one: 一个 dataset 在转换之后还是一个 dataset, 而且 dataset 的 size 不变, 如 Map; 或者一个 dataset 在转换之后还是一个 dataset, 但 size 发生更改, 这种更改有两种可能, 即增大或缩小, 如 flatMap 是 size 增大的操作, 而 subtract 是 size 缩小的操作。
- many-one: 多个 dataset 合并为一个 dataset, 如 Combine、Join。
- one-many: 一个 dataset 分裂为多个 dataset, 如 groupBy。

⑥ Task 运行期间的函数调用

前面介绍了 Task 的提交过程, 下面主要讲解 Task 在运行期间是如何一步步调用作用于 RDD 上的各个 Operation 的。

```
TaskRunner.run
  Task.run
    Task.runTask (Task 是一个基类, 有两个子类, 分别为 ShuffleMapTask 和 ResultTask)
      RDD.iterator
        RDD.computeOrReadCheckpoint
          RDD.compute
```

或许当读者看到 RDD.compute 函数的定义时, 仍觉得 f 没有被调用。以 MappedRDD 的 compute 函数定义为例, 如下:

```
override def compute(split: Partition, context: TaskContext) =
  firstParent[T].iterator(split, context).map(f)
```

注意, 这里最容易产生错觉的地方就是 map 函数。这里的 map 不是 RDD 中的 map, 而是 Scala 中定义的 Iterator 的成员函数 map, 请自行参考 <http://www.scala-lang.org/api/2.10.4/index.html#scala.collection.Iterator>。

⑦ 堆栈输出

```
80 at org.apache.spark.rdd.HadoopRDD.getJobConf (HadoopRDD.scala:111)
81 at org.apache.spark.rdd.HadoopRDD$$anon$1.(HadoopRDD.scala:154)
82 at org.apache.spark.rdd.HadoopRDD.compute (HadoopRDD.scala:149)
83 at org.apache.spark.rdd.HadoopRDD.compute (HadoopRDD.scala:64)
84 at org.apache.spark.rdd.RDD.computeOrReadCheckpoint (RDD.scala:241)
85 at org.apache.spark.rdd.RDD.iterator (RDD.scala:232)
86 at org.apache.spark.rdd.MappedRDD.compute (MappedRDD.scala:31)
87 at org.apache.spark.rdd.RDD.computeOrReadCheckpoint (RDD.scala:241)
88 at org.apache.spark.rdd.RDD.iterator (RDD.scala:232)
89 at org.apache.spark.rdd.FlatMappedRDD.compute (FlatMappedRDD.scala:33)
90 at org.apache.spark.rdd.RDD.computeOrReadCheckpoint (RDD.scala:241)
91 at org.apache.spark.rdd.RDD.iterator (RDD.scala:232)
92 at org.apache.spark.rdd.MappedRDD.compute (MappedRDD.scala:31)
93 at org.apache.spark.rdd.RDD.computeOrReadCheckpoint (RDD.scala:241)
94 at org.apache.spark.rdd.RDD.iterator (RDD.scala:232)
95 at org.apache.spark.rdd.MapPartitionsRDD.compute (MapPartitionsRDD.scala:34)
96 at org.apache.spark.rdd.RDD.computeOrReadCheckpoint (RDD.scala:241)
97 at org.apache.spark.rdd.RDD.iterator (RDD.scala:232)
98 at org.apache.spark.scheduler.ShuffleMapTask.runTask (ShuffleMapTask.
scala:161)
```

```

99    at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.
scala:102)
100   at org.apache.spark.scheduler.Task.run(Task.scala:53)
101   at org.apache.spark.executor.Executor$TaskRunner$$anonfun$run$1.apply
$mcV$sp(Executor.scala:211)

```

⑧ ResultTask

compute 函数的计算过程对于 ShuffleMapTask 来说比较复杂，而对于 ResultTask 来说就简单多了。如下：

```

override def runTask(context: TaskContext): U = {
  metrics = Some(context.taskMetrics)
  try {
    func(context, rdd.iterator(split, context))
  } finally {
    context.executeOnCompleteCallbacks()
  }
}

```

⑨ 计算结果的传递

从上面的分析过程可知，WordCount 这个 Job 在最终提交后，被 DAGScheduler 分为两个 Stage：ShuffleMapTask 和 ResultTask。那么，ShuffleMapTask 的计算结果是如何被 ResultTask 获取的呢？这一过程简述如下：

- ShuffleMapTask 将计算的状态（而不是具体的数据）包装为 MapStatus 返回给 DAGScheduler。
- DAGScheduler 将 MapStatus 保存到 MapOutputTrackerMaster 中。
- ResultTask 在执行到 ShuffleRDD 时会调用 BlockStoreShuffleFetcher 的 fetch 方法获取数据。
- 咨询 MapOutputTrackerMaster 所要获取的数据的位置。
- 根据返回的结果调用 BlockManager.getMultiple 获取真正的数据。

BlockStoreShuffleFetcher 的 fetch 函数伪码如下：

```

val blockManager = SparkEnv.get.blockManager
val startTime = System.currentTimeMillis
val statuses = SparkEnv.get.mapOutputTracker.getServerStatuses(shuffleId,
reduceId)
logDebug("Fetching map output location for shuffle %d, reduce %d took %d
ms".format(
  shuffleId, reduceId, System.currentTimeMillis - startTime))
val blockFetcherItr = blockManager.getMultiple(blocksByAddress, serializer)
val itr = blockFetcherItr.flatMap(unpackBlock)

```

注意上述代码中的 getServerStatuses 和 getMultiple，一个用来询问数据的位置，另一个用来获取真正的数据。

⑩ 两种调度算法的优先级比较^①

调度算法的基本工作之一就是比较两个可执行的 Task 的优先级。Spark 提供的 FIFO 和 FAIR 的

^① 参考 <http://www.cnblogs.com/zwCHAN/p/4246920.html>。

优先级比较在 SchedulingAlgorithm 接口中体现。

FIFO:

- 计算优先级的差。注意，在程序中，通常情况下优先级的数字越小，优先级越高。
- 如果优先级相同，那么 Stage 编号越靠前，优先级越高。
- 如果优先级字段和 Stage 编号都相同，那么 s2 比 s1 优先级高。

FAIR:

- 没有达到最小资源的 Task 比已经达到最小资源的 Task 优先级高。
- 如果两个 Task 都没有达到最小资源，那么比较它们占用最小资源的比例，比例越小，优先级越高；否则比较占用权重资源的比例，比例越小，优先级越高。
- 如果上述比较都相同，那么名字小的优先级高。
- 如果名字相同，则 s2 优先级高。

示例代码如下：

```
/**
 * An interface for sort algorithm
 * FIFO: FIFO algorithm between TaskSetManagers
 * FS: FS algorithm between Pools, and FIFO or FS within Pools
 */
private[spark] trait SchedulingAlgorithm {
  def comparator(s1: Schedulable, s2: Schedulable): Boolean
}

private[spark] class FIFOSchedulingAlgorithm extends SchedulingAlgorithm {
  override def comparator(s1: Schedulable, s2: Schedulable): Boolean = {
    val priority1 = s1.priority
    val priority2 = s2.priority
    var res = math.signum(priority1 - priority2) --计算优先级的差。注意，在程序中，
    通常情况下优先级的数字越小，优先级越高
    if (res == 0) {
      val stageId1 = s1.stageId
      val stageId2 = s2.stageId
      res = math.signum(stageId1 - stageId2) --如果优先级相同，那么 Stage 编号越靠前，
      优先级越高
    }
    if (res < 0) {
      true
    } else {
      false --如果优先级字段和 Stage 编号都相同，那么 s2 比 s1 优先级高
    }
  }
}

private[spark] class FairSchedulingAlgorithm extends SchedulingAlgorithm {
  override def comparator(s1: Schedulable, s2: Schedulable): Boolean = {
    val minShare1 = s1.minShare
    val minShare2 = s2.minShare
```

```

val runningTasks1 = s1.runningTasks
val runningTasks2 = s2.runningTasks
val s1Needy = runningTasks1 < minShare1
val s2Needy = runningTasks2 < minShare2
val minShareRatio1 = runningTasks1.toDouble / math.max(minShare1,1.0).
toDouble
val minShareRatio2 = runningTasks2.toDouble / math.max(minShare2,1.0).
toDouble
val taskToWeightRatio1 = runningTasks1.toDouble / s1.weight.toDouble
val taskToWeightRatio2 = runningTasks2.toDouble / s2.weight.toDouble
var compare:Int = 0
if (s1Needy && !s2Needy) {
    return true
} else if (!s1Needy && s2Needy) {    --没有达到最小资源的 Task 比已经达到最小资源的
Task 优先级高
    return false
} else if (s1Needy && s2Needy) {    --如果两个 Task 都没有达到最小资源，那么比较它
们占用最小资源的比例，比例越小，优先级越高
    compare = minShareRatio1.compareTo(minShareRatio2)
} else {    --否则比较占用权重资源的比例，比例越小，优先级越高
    compare = taskToWeightRatio1.compareTo(taskToWeightRatio2)
}
if (compare < 0) {
    true
} else if (compare > 0) {
    false
} else {--如果上述比较都相同，那么名字小的优先级高；如果名字相同，则 s2 优先级高
    s1.name < s2.name
}
}
}

```

7) Shuffle 实现^①

在 MapReduce 框架中，Shuffle 是连接 Map 和 Reduce 的桥梁，在 Map 和 Reduce 两个过程中必须经过 Shuffle 这个环节，Shuffle 的性能高低直接影响到整个程序的性能和吞吐量。Spark 作为 MapReduce 框架的一种实现，自然也就实现了 Shuffle 的逻辑。下面深入研究 Spark 的 Shuffle 是如何实现的、有什么优缺点，以及与 Hadoop MapReduce 的 Shuffle 有什么不同。

① Shuffle 简介

Shuffle 是 MapReduce 框架中一个特定的 Phase，介于 Map Phase 和 Reduce Phase 之间。当 Map 的输出结果被 Reduce 使用时，输出结果需要按 Key 进行哈希运算，并且分发到每一个 Reducer 上，这个过程就是 Shuffle。由于 Shuffle 涉及磁盘的读/写和网络的传输，因此 Shuffle 性能的高低直接影响到整个程序的运行效率。

^① 参考 <http://jerryshao.me/architecture/2014/01/04/spark-shuffle-detail-investigation/>。

图 7.16 清晰地描述了 MapReduce 算法的整个流程。

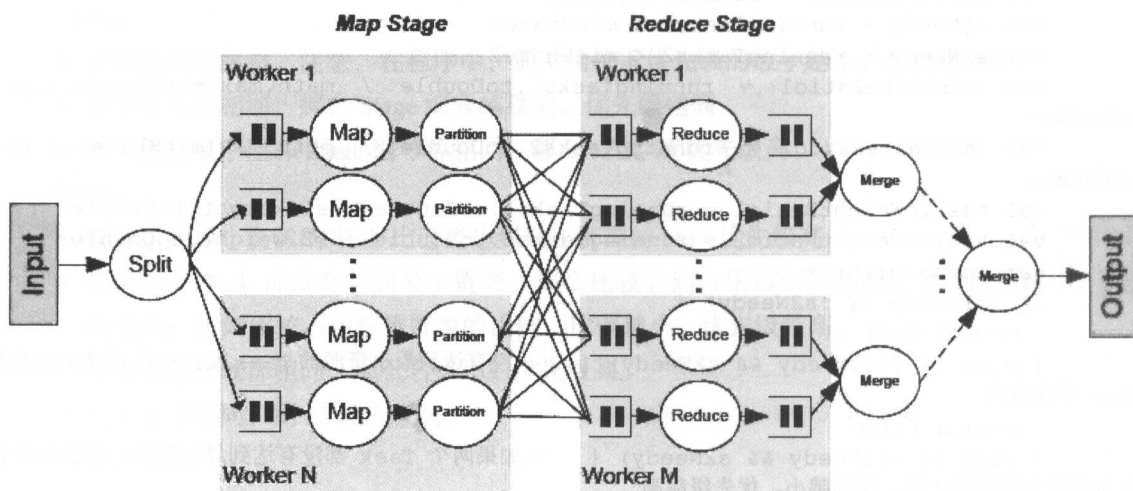


图 7.16

概念上, Shuffle 就是一座沟通数据连接的桥梁; 那么实际上, Shuffle 这一部分是如何实现的呢? 下面就以 Spark 为例, 讲解 Shuffle 在 Spark 中的实现。

② Spark Shuffle 进化史

先以图为例来简单描述一下 Spark 中 Shuffle 的整个流程, 如图 7.17 所示。

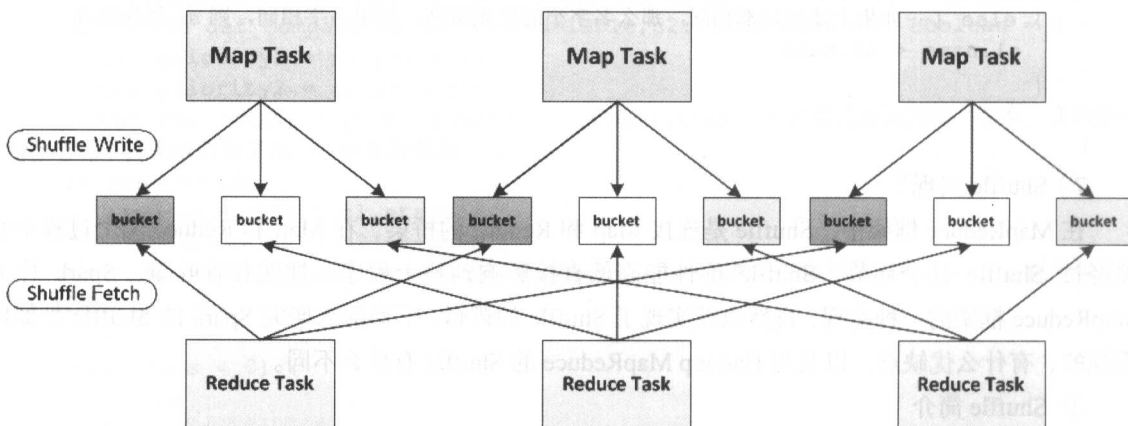


图 7.17

首先, 每个 Mapper 会根据 Reducer 的数量创建相应的 bucket, bucket 的数量是 $M \times R$, 其中 M 是 Map 的个数, R 是 Reduce 的个数。

其次, Mapper 产生的结果会根据设置的 Partition 算法填充到每个 bucket 中去。这里的 Partition 算法是可以自定义的, 当然, 默认算法是根据 Key 哈希到不同的 bucket 中去。

最后，当 Reducer 启动时，它会根据自己 Task 的 ID 和所依赖的 Mapper 的 ID 从远端或本地的 Block Manager 中取得相应的 bucket，作为 Reducer 的输入进行处理。

这里的 bucket 是一个抽象概念，在实现中，每个 bucket 可以对应一个文件，也可以对应文件的一部分或其他。

接下来分别从 Shuffle Write 和 Shuffle Fetch 两个方面来讲述一下 Spark 的 Shuffle 进化史。

(1) Shuffle Write。

在 Spark 0.6 和 0.7 两个版本中，将 Shuffle 数据以文件形式存储在 Block Manager 中，与 `rdd.persist(StorageLevel.DISK_ONLY)` 采取的策略相同，参看如下代码：

```
override def run(attemptId: Long): MapStatus = {
  val numOutputSplits = dep.partitioner.numPartitions
  ...
  // Partition the map output.
  val buckets = Array.fill(numOutputSplits)(new ArrayBuffer[(Any,Any)])
  for (elem <- rdd.iterator(split,taskContext)) {
    val pair = elem.asInstanceOf[(Any,Any)]
    val bucketId = dep.partitioner.getPartition(pair._1)
    buckets(bucketId) += pair
  }
  ...
  val blockManager = SparkEnv.get.blockManager
  for (i <- 0 until numOutputSplits) {
    val blockId = "shuffle_" + dep.shuffleId + "_" + partition + "_" + i
    // Get a Scala iterator from Java map
    val iter: Iterator[(Any,Any)] = buckets(i).iterator
    val size = blockManager.put(blockId,iter,StorageLevel.DISK_ONLY,false)
    totalBytes += size
  }
  ...
}
```

从上述代码中可以看到，Spark 在每个 Mapper 中为每个 Reducer 创建一个 bucket，并将 RDD 的计算结果放到 bucket 中。需要注意的是，每个 bucket 是一个 `ArrayBuffer`，也就是说 Map 的输出结果会先存储在内存中。

接着，Spark 会将 `ArrayBuffer` 中的 Map 输出结果写入 Block Manager 所管理的磁盘中，这里文件的命名方式为：`shuffle_ + shuffle_id + "_" + map partition id + "_" + shuffle partition id`。

早期的 Shuffle Write 有两个比较大的问题：

- Map 的输出必须先全部存储到内存中，然后才能写入磁盘。这对于内存来说是一个非常大的开销，当内存不足时，就会出现 OOM。
- 每个 Mapper 都会产生 Reducer number 个 Shuffle 文件，如果 Mapper 的个数是 1k，Reducer 的个数也是 1k，那么就会产生 1M 个 Shuffle 文件，这对于文件系统来说是一个非常大的负担。同时在 Shuffle 数据量不大而 Shuffle 文件又非常多的情况下，随机写也会严重降低 I/O 的性能。

在 Spark 0.8 版本中, Shuffle Write 采用了与 RDD Block Write 不同的方式, 同时也为 Shuffle Write 单独创建了 ShuffleBlockManager, 部分解决了 0.6 和 0.7 版本中遇到的问题。

首先来看一下 Spark 0.8 版本的具体实现, 如下:

```
override def run(attemptId: Long): MapStatus = {
  ...
  val blockManager = SparkEnv.get.blockManager
  var shuffle: ShuffleBlocks = null
  var buckets: ShuffleWriterGroup = null
  try {
    // Obtain all the block writers for shuffle blocks.
    val ser = SparkEnv.get.serializerManager.get(dep.serializerClass)
    shuffle = blockManager.shuffleBlockManager.forShuffle (dep.shuffleId,
numOutputSplits, ser)
    buckets = shuffle.acquireWriters(partition)
    // Write the map output to its associated buckets.
    for (elem <- rdd.iterator(split, taskContext)) {
      val pair = elem.asInstanceOf[Product2[Any, Any]]
      val bucketId = dep.partitionner.getPartition(pair._1)
      buckets.writers(bucketId).write(pair)
    }
    // Commit the writes. Get the size of each bucket block (total block size).
    var totalBytes = 0L
    val compressedSizes: Array[Byte] = buckets.writers.map { writer: BlockObject
Writer =>
      writer.commit()
      writer.close()
      val size = writer.size()
      totalBytes += size
      MapOutputTracker.compressSize(size)
    }
    ...
  } catch { case e: Exception =>
    // If there is an exception from running the task, revert the partial writes
    // and throw the exception upstream to Spark.
    if (buckets != null) {
      buckets.writers.foreach(_._revertPartialWrites())
    }
    throw e
  } finally {
    // Release the writers back to the shuffle block manager.
    if (shuffle != null && buckets != null) {
      shuffle.releaseWriters(buckets)
    }
    // Execute the callbacks on task completion.
    taskContext.executeOnCompleteCallbacks()
  }
}
```

在 Spark 0.8 版本中为 Shuffle Write 添加了一个新的类 `ShuffleBlockManager`，由该类负责分配和管理 bucket。同时 `ShuffleBlockManager` 为每个 bucket 分配一个 `DiskObjectWriter`，每个 Write Handler 默认拥有 100KB 的缓存，可以使用这个 Write Handler 将 Map output 写入文件中。可以看到，现在的写入方式变为 `buckets.writers(bucketId).write(pair)`，也就是说 Map output 的 key-value pair 是逐个写入磁盘中的，而不是预先把所有数据存储在内存中再整体写到磁盘中。

`ShuffleBlockManager` 的代码如下：

```
private[spark]
class ShuffleBlockManager(blockManager: BlockManager) {
  def forShuffle(shuffleId: Int, numBuckets: Int, serializer: Serializer):
  ShuffleBlocks = {
    new ShuffleBlocks {
      // Get a group of writers for a map task.
      override def acquireWriters(mapId: Int): ShuffleWriterGroup = {
        val bufferSize = System.getProperty("spark.shuffle.file.buffer.kb",
"100").toInt * 1024
        val writers = Array.tabulate[BlockObjectWriter](numBuckets) { bucketId =>
          val blockId = ShuffleBlockManager.blockId(shuffleId, bucketId, mapId)
          blockManager.getDiskBlockWriter(blockId, serializer, bufferSize)
        }
        new ShuffleWriterGroup(mapId, writers)
      }
      override def releaseWriters(group: ShuffleWriterGroup) = {
        // Nothing really to release here.
      }
    }
  }
}
```

Spark 0.8 显著减少了 Shuffle 的内存压力，现在 Map output 不需要先全部存储在内存中，再写到磁盘中，而是按照记录级别写入磁盘中。同时，对于 Shuffle 文件，也独立出新的 `ShuffleBlockManager` 进行管理。

但是 Spark 0.8 版本的 Shuffle Write 仍然有两个大的问题没有解决。

- 首先是 Shuffle 文件过多的问题。Shuffle 文件过多，一方面会造成文件系统的压力过大，另一方面会降低 I/O 的吞吐量。
- 其次，虽然 Map output 数据不需要预先存储在内存中，从而显著减少了内存压力，但是新引入的 `DiskObjectWriter` 所带来的 buffer 开销也不容小觑。假定我们有 1k 个 Mapper 和 1k 个 Reducer，那么就会有 1M 个 bucket，与此同时就会有 1M 个 Write Handler，而每个 Write Handler 默认需要 100KB 内存，那么共需要 100GB 内存。当然，实际情况下这 1k 个 Mapper 是分时运行的，所需的内存只有 $\text{cores} \times \text{reducer numbers} \times 100\text{KB}$ 。但是，如果 Reducer 的数量很多，那么这个 buffer 的内存开销也是很高的。

为了解决 Shuffle 文件过多的情况，Spark 0.8.1 引入了新的 Shuffle Consolidation，以期显著减少 Shuffle 文件的数量。

首先以图例来介绍一下 Shuffle Consolidation 的原理，如图 7.18 所示。

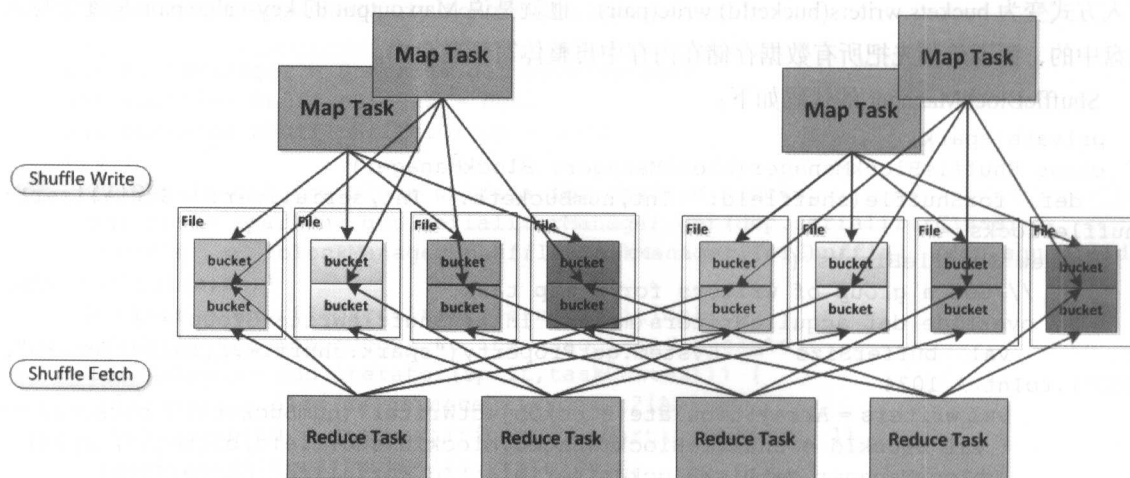


图 7.18

假定该 Job 有 4 个 Mapper、4 个 Reducer 和 2 个 core，也就是能并行运行两个 Task。由此可以计算出 Spark 的 Shuffle Write 共需要 16 个 bucket，也就有了 16 个 Write Handler。在之前的 Spark 版本中，每个 bucket 对应一个文件，因此，在这里会产生 16 个 Shuffle 文件。

而在 Shuffle Consolidation 中，每个 bucket 并非对应一个文件，而是对应文件中的一个 Segment，同时 Shuffle Consolidation 所产生的 Shuffle 文件数量与 Spark core 的个数也有关系。在图 7.18 中，Job 的 4 个 Mapper 分两批运行，在第一批的两个 Mapper 运行时，会申请 8 个 bucket，产生 8 个 Shuffle 文件；而在第二批的两个 Mapper 运行时，申请的 8 个 bucket 并不会再产生 8 个新的文件，而是追加到之前的 8 个文件后面，这样一来就只有 8 个 Shuffle 文件，而在文件内部只有 16 个不同的 Segment。因此，从理论上讲，Shuffle Consolidation 所产生的 Shuffle 文件数量为 $C \times R$ ，其中 C 是 Spark 集群的 core 个数， R 是 Reducer 的个数。

需要注意的是，当 $M=C$ 时，Shuffle Consolidation 所产生的文件数量与之前的实现相同。

Shuffle Consolidation 显著减少了 Shuffle 文件的数量，解决了之前版本存在的一个比较严重的问题，但是 Writer Handler 的 buffer 开销依然过大。若要减少 Writer Handler 的 buffer 开销，只能减少 Reducer 的数量，但是这又会引入新的问题，下文将详细介绍。

(2) Shuffle Fetch and Aggregator.

Shuffle Write 写出去的数据要被 Reducer 使用，就需要 Shuffle Fetcher 将所需的数据取过来。Spark 对 Shuffle Fetcher 实现了两套不同的框架：NIO 通过 Socket 连接去取数据；OIO 通过 Netty Server 去取数据。这两套框架分别对应 BasicBlockFetcherIterator 和 NettyBlockFetcherIterator 类。

在 Spark 0.7 及早期版本中, 只支持 BasicBlockFetcherIterator, 而 BasicBlockFetcherIterator 在 Shuffle 数据量比较大的情况下表现始终不是很好, 无法充分利用网络带宽。为了解决这个问题, 添加了新的 Shuffle Fetcher 来试图获得更好的性能。

接下来说一下 Aggregator。在 Hadoop MapReduce 的 Shuffle 过程中, Shuffle 获取过来的数据会进行 merge sort (合并排序), 使得相同 Key 下的不同 Value 按序归并到一起供 Reducer 使用, 这个过程可以参看图 7.19。

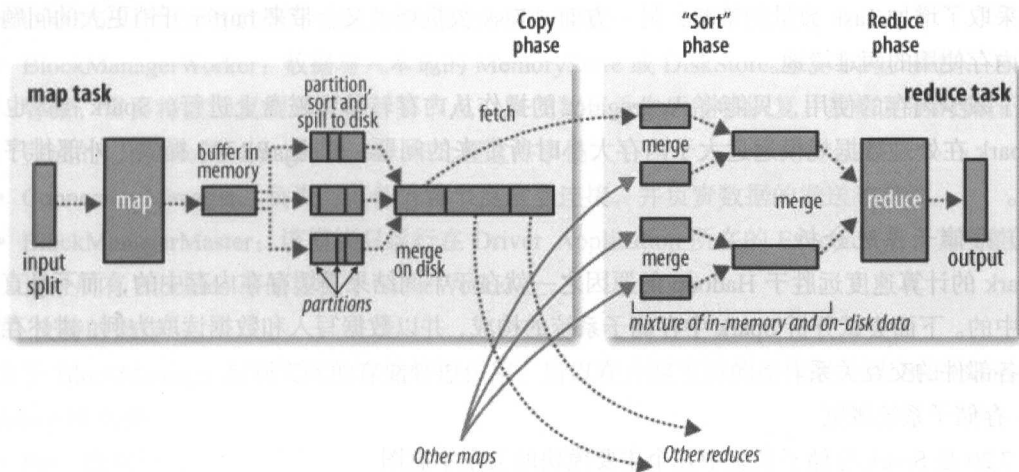


图 7.19

所有的 merge sort 都是在磁盘上进行的, 从而有效地控制了内存的使用, 但代价是更多的磁盘 I/O。

那么, Spark 是否也有 merge sort, 还是以其他方式实现? 下面来详细说明。

虽然 Spark 属于 MapReduce 体系, 但是对传统的 MapReduce 算法进行了一定的改变。Spark 假定在大多数用户的 case 中, Shuffle 数据的 sort 不是必需的, 强制进行排序只会使性能变差, 因此 Spark 并不在 Reducer 端进行 merge sort。既然没有 merge sort, 那么 Spark 是如何进行 Reduce 的呢? 这就要涉及 Aggregator 了。

Aggregator 本质上是一个 Hash map, 它是以 map output 的 Key 为 Key、以任意所要合并的类型为 Value 的 Hash map。当 word count 这个计算进行 reduce 计算 count 值的时候, 它会将 Shuffle 获取到的每个 Key/Value 对更新或插入到 Hash map 中。这样就不需要预先把所有的 Key/Value 对进行 merge sort, 而是来一个处理一个, 省去了外部排序这一步骤。需要注意的是, Reducer 的内存必须足以存放这个 Partition 的所有 Key 和 Count 值, 因而对内存有一定的要求。

在上面的例子中, 因为 Value 会不断地更新, 不需要将其全部记录在内存中, 因而内存的使用还是比较少的。考虑一下, 如果是按 Key 分组这样的操作, Reducer 需要得到 Key 对应的所有 Value。在 Hadoop MapReduce 中, 由于有了 merge sort, 因此给予 Reducer 的数据已经按 Key 分组了, 而 Spark 中没有这一步操作, 因此需要将 Key 和对应的 Value 全部存放在 Hash map 中, 并将 Value 合并成一

个 Array。为了能够存放所有的数据，用户必须确保每个 Partition 足够小，这对内存而言是一个严峻的考验。因此，Spark 文档中建议用户涉及这类操作时尽量增加 Partition，也就是增加 Mapper 和 Reducer 的数量。

增加 Mapper 和 Reducer 的数量固然可以减小 Partition，使得内存可以容纳这个 Partition，但是 bucket 和对应于 bucket 的 Write Handler 是由 Mapper 和 Reducer 的数量决定的，Task 越多，bucket 就会增加得更多，从而导致 Write Handler 所需的 buffer 也会更多。这样，一方面，为了减少内存的使用而采取了增加 Task 数量的策略；另一方面，Task 数量增多又会带来 buffer 开销更大的问题，从而陷入内存使用的两难境地。

为了减少内存的使用，只能将 Aggregator 的操作从内存转移到磁盘上进行，Spark 社区也意识到了 Spark 在处理数据规模远远大于内存大小时所带来的问题，因此，PR303 提供了外部排序的实现方案。

8) 存储子系统分析

Spark 的计算速度远胜于 Hadoop 的原因之一就在于中间结果是缓存在内存中的，而不是直接写入磁盘中的。下面尝试分析 Spark 中存储子系统的构成，并以数据写入和数据读取为例，讲述存储子系统中各部件的交互关系。

① 存储子系统概览

图 7.20 是 Spark 存储子系统中几个主要模块的关系示意图。

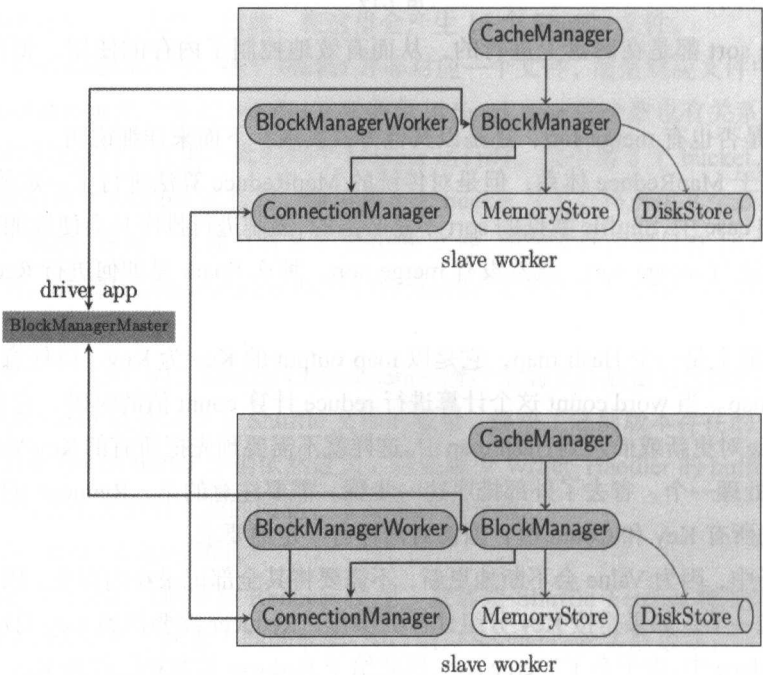


图 7.20

下面简要介绍其中的几个主要模块。

- **CacheManager**: RDD 在进行计算的时候,通过 **CacheManager** 来获取数据,并通过 **CacheManager** 来存储计算结果。
- **BlockManager**: **CacheManager** 在进行数据读取和存储的时候,主要依赖 **BlockManager** 接口来实现, **BlockManager** 决定数据是从内存 (**MemoryStore**) 还是从磁盘 (**DiskStore**) 中获取。
- **MemoryStore**: 负责将数据保存在内存中或从内存中读取。
- **DiskStore**: 负责将数据写入磁盘中或从磁盘中读取。
- **BlockManagerWorker**: 数据写入本地的 **MemoryStore** 或 **DiskStore** 是一个同步操作,为了实现容错,需要将数据复制到其他计算节点,以便数据丢失后能够恢复。数据复制的操作是异步完成的,由 **BlockManagerWorker** 来处理这件事情。
- **ConnectionManager**: 负责与其他计算节点建立连接,并负责数据的发送和接收。
- **BlockManagerMaster**: 该模块只运行在 **Driver Application** 所在的 **Executor** 上,其功能是负责记录所有 **BlockIds** 存储在哪个 **SlaveWorker** 上。

② 支持的操作

由于 **BlockManager** 起到实际的存储管控作用,所以在介绍支持的操作时,以 **BlockManager** 中的 **Public API** 为例。

- **Put**: 数据写入。
- **Get**: 数据读取。
- **remoteRDD**: 数据删除。一旦整个 **Job** 完成,所有的中间计算结果都可以删除。

③ 启动过程分析

上述各个模块由 **SparkEnv** 来创建,创建过程在 **SparkEnv.create** 中完成。

```
val blockManagerMaster = new BlockManagerMaster(registerOrLookup(
    "BlockManagerMaster",
    new BlockManagerMasterActor(isLocal, conf)), conf)
val blockManager = new BlockManager(executorId, actorSystem, blockManagerMaster,
    serializer, conf)
val connectionManager = blockManager.connectionManager
val broadcastManager = new BroadcastManager(isDriver, conf)
val cacheManager = new CacheManager(blockManager)
```

这段代码很容易让人疑惑,看起来像在所有的 **Cluster** 节点上都创建了 **BlockManagerMasterActor**。

其实不然,仔细看 **registerOrLookup** 函数的实现。如果当前节点是 **Driver**,则创建这个 **Actor**; 否则建立到 **Driver** 的连接。

```
def registerOrLookup(name: String, newActor: => Actor): ActorRef = {
    if (isDriver) {
        logInfo("Registering " + name)
        actorSystem.actorOf(Props(newActor), name = name)
    } else {
        val driverHost: String = conf.get("spark.driver.host", "localhost")
```

```

val driverPort: Int = conf.getInt("spark.driver.port", 7077)
Utils.checkHost(driverHost, "Expected hostname")
val url = s"akka.tcp://spark@$driverHost:$driverPort/user/$name"
val timeout = AkkaUtils.lookupTimeout(conf)
logInfo(s"Connecting to $name: $url")
Await.result(actorSystem.actorSelection(url).resolveOne(timeout), timeout)
}
}

```

初始化过程中的一个主要动作就是 BlockManager 需要向 BlockManagerMaster 发起注册。

④ 数据写入过程分析

数据写入的简要流程如图 7.21 所示。

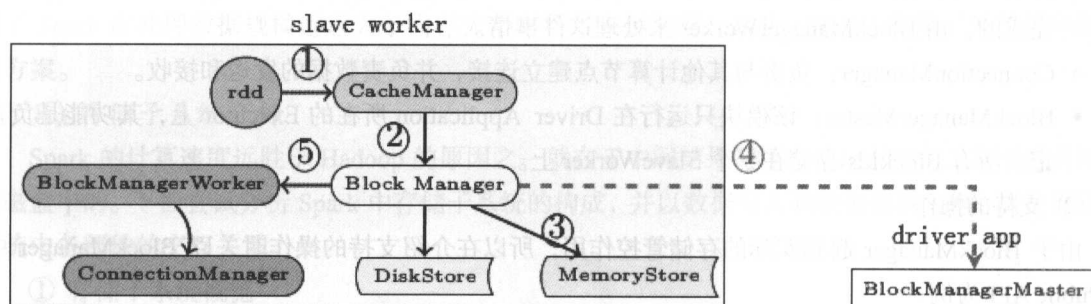


图 7.21

(1) RDD.iterator 是与存储子系统交互的入口。

(2) CacheManager.getOrCompute 调用 BlockManager 的 put 接口来写入数据。

(3) 数据优先写入 MemoryStore (内存)。如果 MemoryStore 中的数据已满，则将最近使用次数不频繁的数据写入磁盘。

(4) 通知 BlockManagerMaster 有新的数据写入，在 BlockManagerMaster 中保存元数据。

(5) 将写入的数据与其他 Slave Worker 进行同步。一般来说，在本机写入的数据，同时会用另一台机器来进行备份。

⑤ 序列化与否

写入的具体内容可以是序列化之后的 Bytes，也可以是没有序列化的 Value。此处有一个对 Scala 的语法中 Either、Left、Right 关键字的理解。

⑥ 数据读取过程分析

数据读取过程的代码如下：

```

def get(blockId: BlockId): Option[Iterator[Any]] = {
  val local = getLocal(blockId)
  if (local.isDefined) {
    logInfo("Found block %s locally".format(blockId))
    return local
  }
  val remote = getRemote(blockId)
}

```

```

if (remote.isDefined) {
  logInfo("Found block %s remotely".format(blockId))
  return remote
}
None
}

```

⑦ 本地读取

首先查询在本机的 MemoryStore 和 DiskStore 中是否存在所需的 Block 数据，如果不存在，则发起远程数据获取。

⑧ 远程读取

远程获取调用路径：`getRemote` → `doGetRemote`。在 `doGetRemote` 中最主要的就是调用 `BlockManagerWorker.syncGetBlock` 来远程获取数据。代码如下：

```

def syncGetBlock(msg: GetBlock, toConnManagerId: ConnectionManagerId):
ByteBuffer = {
  val blockManager = blockManagerWorker.blockManager
  val connectionManager = blockManager.connectionManager
  val blockMessage = BlockMessage.fromGetBlock(msg)
  val blockMessageArray = new BlockMessageArray(blockMessage)
  val responseMessage = connectionManager.sendMessageReliablySync(
    toConnManagerId, blockMessageArray.toBufferMessage)
  responseMessage match {
    case Some(message) => {
      val bufferMessage = message.asInstanceOf[BufferMessage]
      logDebug("Response message received " + bufferMessage)
      BlockMessageArray.fromBufferMessage(bufferMessage).foreach
(blockMessage => {
        logDebug("Found " + blockMessage)
        return blockMessage.getData
      })
    }
    case None => logDebug("No response message received")
  }
  null
}

```

在上述代码中，最有意思的莫过于 `sendMessageReliablySync`。远程数据读取毫无疑问是一个异步 I/O 操作，这里的代码为什么写起来就像在进行同步操作呢？

别急，继续看看 `sendMessageReliablySync` 的定义。

```

def sendMessageReliably(connectionManagerId: ConnectionManagerId, message: Message)
: Future[Option[Message]] = {
  val promise = Promise[Option[Message]]
  val status = new MessageStatus(
    message, connectionManagerId, s => promise.success(s.ackMessage))
  messageStatuses.synchronized {
    messageStatuses += ((message.id, status))
  }
}

```

```

    }
    sendMessage(connectionManagerId,message)
    promise.future
  }

```

注意此处的关键字 `promise` 和 `future`。

如果这个 `future` 执行完毕，则返回 `s.ackMessage`。再来看看这个 `ackMessage` 是在什么地方被写入的。看看 `ConnectionManager.handleMessage` 中的代码片段：

```

case bufferMessage: BufferMessage => {
  if (authEnabled) {
    val res = handleAuthentication(connection,bufferMessage)
    if (res == true) {
      // message was security negotiation so skip the rest
      logDebug("After handleAuth result was true,returning")
      return
    }
  }
  if (bufferMessage.hasAckId) {
    val sentMessageStatus = messageStatuses.synchronized {
      messageStatuses.get(bufferMessage.ackId) match {
        case Some(status) => {
          messageStatuses -= bufferMessage.ackId
          status
        }
        case None => {
          throw new Exception("Could not find reference for received ack
message " +
          message.id)
          null
        }
      }
    }
    sentMessageStatus.synchronized {
      sentMessageStatus.ackMessage = Some(message)
      sentMessageStatus.attempted = true
      sentMessageStatus.acked = true
      sentMessageStatus.markDone()
    }
  }
}

```

注意，此处调用的 `sentMessageStatus.markDone` 方法会调用在 `sendMessageReliablySync` 中定义的 `promise.Success`，具体可以看看 `MessageStatus` 的定义。

```

class MessageStatus(
  val message: Message,
  val connectionManagerId: ConnectionManagerId,
  completionHandler: MessageStatus => Unit) {
  var ackMessage: Option[Message] = None
  var attempted = false
  var acked = false
}

```

```
def markDone() { completionHandler(this) }
}
```

至此，调用关系全部搞清楚了。

⑨ TachyonStore

在 Spark 的最新源码中，存储子系统引入了 TachyonStore。TachyonStore 是在内存中实现了 HDFS 文件系统的接口，其主要目的是尽可能地利用内存来作为数据持久层，避免过多的磁盘 I/O 操作。

3. Spark 2.0 的主要特性

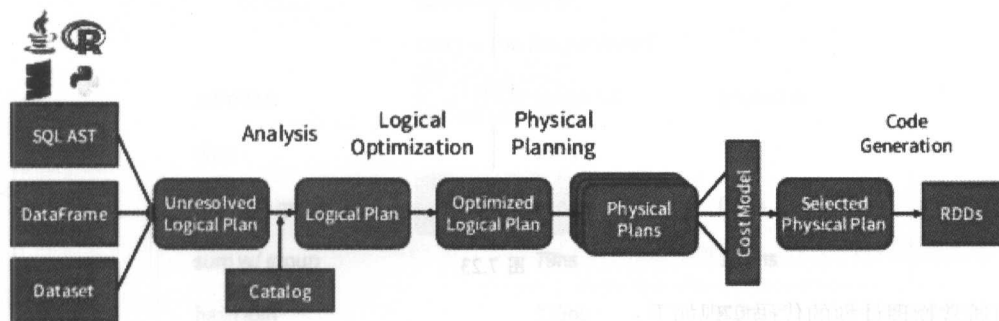
相比之前的版本，Spark 2.0 有很大的性能提升。

1) 统一 API 到 Dataset

DataFrame 和 Dataset 都是提供给用户使用的，包括各类操作接口的 API。1.3 版本引入 DataFrame，1.6 版本引入 Dataset，2.0 版本将二者统一，即保留 Dataset，而把 DataFrame 定义为 Dataset[Row]，即 Dataset 里的元素对象为 Row 的一种（SPARK-13485）。

DataFrame 提供了一系列操作 API，与 RDD API 相比，DataFrame 里操作的数据都带有 Schema 信息，所以 DataFrame 里的所有操作可以享受 Spark SQL Catalyst optimizer 带来的性能提升，如 Code Generation、Tungsten 等。执行过程如图 7.22 所示。

Shared Optimization & Execution



DataFrames, Datasets and SQL
share the same optimization/execution pipeline

databricks

图 7.22

但是后来发现，在有些情况下，用 RDD 可以表达的逻辑用 DataFrame 无法表达。比如，要对 group by 或 join 后的结果用自定义的函数，可能用 SQL 是无法表达的。如下代码：

```
case class ClassData(a: String, b: Int)
case class ClassNullableData(a: String, b: Integer)
val ds = Seq(ClassData("a", 1), ClassData("a", 2)).toDS()
val agged = ds.groupByKey(d => ClassNullableData(d.a, null))
.mapGroups {
```

```
case (key, values) => key.a + values.map(_._b).sum
}
```

中间处理过程的数据是自定义的，并且 group by 后的聚合逻辑也是自定义的，故用 SQL 比较难以表达，因而提出了 Dataset API。Dataset API 扩展 DataFrame API，支持静态类型和用户自定义函数。同时，Dataset 也能享受 Spark SQL 里所有性能带来的提升。后来发现 Dataset 包含了 DataFrame 的功能，这样二者就出现了很大的冗余，故在 2.0 版本中将二者统一，保留 Dataset API，把 DataFrame 表示为 Dataset[Row]，即 Dataset 的子集。

因此，在使用 API 时，优先选择 DataFrame & Dataset，但是为了兼容早期版本的程序，RDD API 也会一直保留。后续 Spark 上层的库将全部使用 DataFrame，如 MLlib、Streaming、GraphX 等。

2) 全流程代码自动生成

先看一个例子：

```
select count(*) from store_sales where ss_item_sk = 1000
```

翻译成计算引擎的执行计划如图 7.23 所示。

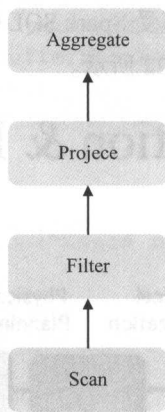


图 7.23

而通常物理计划的代码实现如下：

```
class Filter {
  def next(): Boolean = {
    var found = false
    while (!found && child.next()) {
      found = predicate(child.fetch())
    }
    return found
  }
  def fetch(): InternalRow = {
    child.fetch()
  }...
}
```

但是如果用 hard code 来写，则代码如下：

```

var count = 0
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1
  }
}

```

二者的关系如图 7.24 所示。

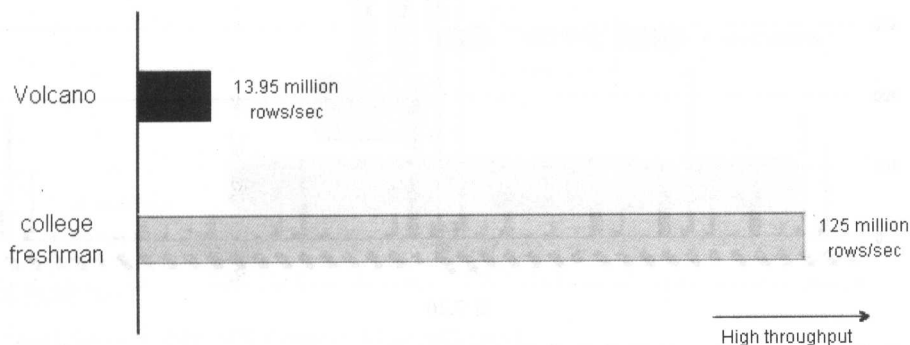


图 7.24

那么，如何使计算引擎的物理执行速度达到 hard code 的性能呢？于是提出了 whole-stage code generation，即对物理执行的多次调用转换为代码 for 循环。此次优化带来的性能提升如图 7.25 所示。

cost per row (single thread)		
primitive	Spark 1.6	Spark 2.0
filter	15ns	1.1ns
sum w/o group	14ns	0.9ns
sum w/ group	79ns	10.7ns
hash join	115ns	4.0ns
sort (8-bit entropy)	620ns	5.3ns
sort (64-bit entropy)	620ns	40ns
sort-merge join	750ns	700ns

图 7.25

从结果可以看出，使用该特性后，各操作的性能都有很大的提升。

TPC-DS 的对比测试结果（Spark 1.6 对比 Spark 2.0）如图 7.26 所示。

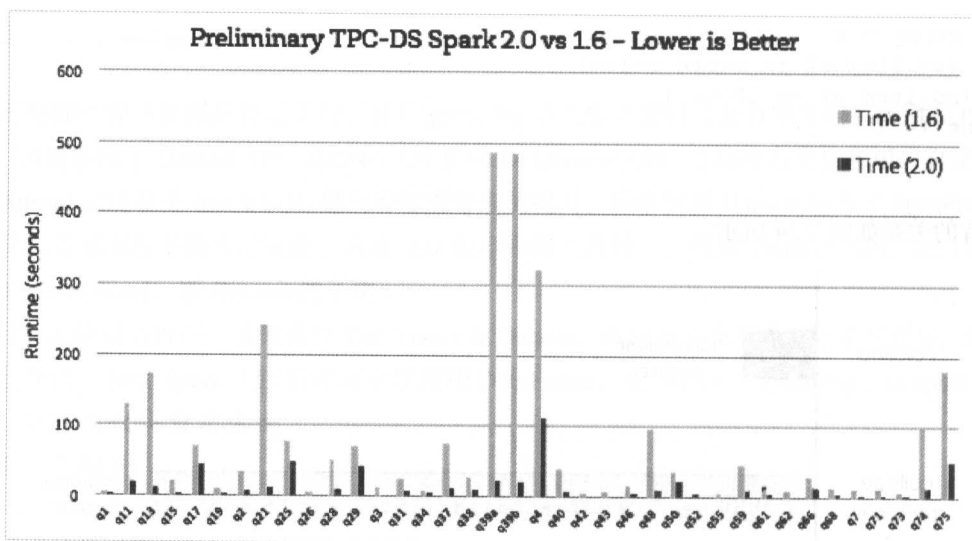


图 7.26

3) 抛弃 DStream API, 新增结构化流 API

Spark Streaming 把流式计算看作一个个离线计算, 提供了一套 DStream 的流 API。相比其他的流式计算, Spark Streaming 的优点是有较高的容错性和吞吐量。关于 Spark Streaming 的详细设计思想和分析, 可以到 <https://github.com/lw-lin/CoolplaySpark> 进行详细学习和了解。

在 2.0 版本之前, 如果既有流式计算, 又有离线计算, 则需要用两套 API 来编写程序, 一套是 RDD API, 另一套是 DStream API。而且 DStream API 在易用性上远不如 SQL 或 DataFrame。

为了真正将流式计算和离线计算在编程 API 上统一, 同时也让 Streaming 作业能够享受 DataFrame/Dataset 所带来的优势, 于是提出了 Structured Streaming。现在只要基于 DataFrame/Dataset 来开发流式计算和离线计算的程序, 即可得到与 Dataflow 同样的效果。

比如, 在做 Batch Aggregation 时, 可以写成如下代码:

```
logs = ctx.read.format("json").open("s3://logs")

logs.groupBy(logs.user_id).agg(sum(logs.time))
    .write.format("jdbc")
    .save("jdbc:mysql/...")
```

那么, 对于流式计算, 仅仅调用了 DataFrame/Dataset 的不同函数代码, 如下:

```
logs = ctx.read.format("json").stream("s3://logs")

logs.groupBy(logs.user_id).agg(sum(logs.time))
    .write.format("jdbc")
    .stream("jdbc:mysql/...")
```

最后, 在 DataFrame/Dataset 这个 API 上可以完成如图 7.27 所示的所有应用。

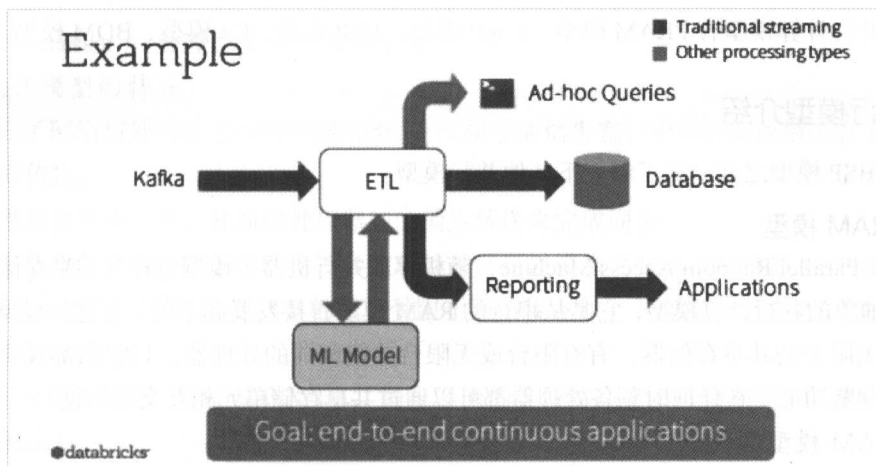


图 7.27

4) 其他特性

- 用 SparkSession 替换 SQLContext 和 HiveContext。
- MLlib 里的计算用 DataFrame-based API 代替 RDD 计算逻辑。
- 提供更多的分布式 R 语言算法。
- 支持 ML Pipeline 持久化。
- 更简单、更高性能的 Accumulator API。

7.5 BSP 框架^①

Spark、Hadoop 是迭代模式，只适合一般的计算，在机器学习等计算量非常大的领域，传统的迭代模型不再适用。本节介绍的 BSP 就是为了解决一些特定场景下的计算量问题。

7.5.1 什么是 BSP 模型

BSP (Bulk Synchronous Parallel, 整体同步并行计算模型) 是一种并行计算模型，由英国计算机科学家 Viliant 在 20 世纪 80 年代提出。Google 发布的一篇论文 (*Pregel: A System for Large-Scale Graph Processing*) 使得这一概念被更多人所认识。和 MapReduce 一样，Google 并没有开源 Pregel，Apache 按 Pregel 的思想提供了类似的框架 Hama。

并行计算模型通常指从并行算法的设计和分析出发，将各种并行计算机（至少是某一类并行计算机）的基本特征抽象出来，形成一个抽象的计算模型。从更广的意义上说，并行计算模型为并行计算提供了硬件和软件界面，在该界面的约定下，并行系统硬件设计者和软件设计者可以开发对并行性的支持机制，从而提高系统的性能。

① 参考 <http://www.cnblogs.com/BYRans/p/4682282.html>。

常用的并行计算模型有 PRAM 模型、LogP 模型、BSP 模型、C3 模型、BDM 模型。

7.5.2 并行模型介绍

在介绍 BSP 模型之前，先了解一下其他并行模型。

1. PRAM 模型

PRAM (Parallel Random Access Machine, 随机存取并行机器) 模型也称为共享存储的 SIMD 模型，是一种抽象的并行计算模型，它是从串行的 RAM 模型直接发展起来的。在这种模型中，假定存在一台容量无限大的共享存储器，有有限台或无限台功能相同的处理器，且它们都具有简单的算术运算和逻辑判断功能，在任何时刻各处理器都可以通过共享存储单元相互交互数据。

1) PRAM 模型的优点

PRAM 模型特别适合并行算法的表达、分析和比较，使用简单，很多关于并行计算机的底层细节，如处理器间通信、存储系统管理和进程同步等，都被隐含在该模型中；易于设计算法和稍加修改便可以运行在不同的并行计算机系统中；根据需要，可以在 PRAM 模型中加入一些诸如同步和通信等需要考虑的内容。

2) PRAM 模型的缺点

(1) 模型中使用了一台全局共享存储器，且局存容量较小，不足以描述分布主存多处理器的性能瓶颈，而且共享单一存储器的假定，显然不适合分布存储结构的 MIMD 机器。

(2) PRAM 模型是同步的，这就意味着所有的指令都按照锁步 (Clock Step) 的方式操作。用户虽然感觉不到同步的存在，但同步的存在的确很耗费时间，而且不能反映现实中很多系统的异步性。

(3) PRAM 模型假设每台处理器可在单位时间内访问共享存储器的任一单元，因此要求处理器间通信无延迟、无限带宽和无开销。假定每台处理器均可以在单位时间内访问任何存储单元而略去了实际存在的、合理的细节，如资源竞争和有限带宽，这是不现实的。

(4) 未能描述多线程技术和流水线预取技术，而这两种技术又是当今并行体系结构应用最普遍的技术。

2. LogP 模型

LogP 模型是由 Culler (1993) 提出的，是一种分布存储的、点到点通信的多处理器模型。其中通信由一组参数描述，实行隐式同步。

LogP 模型的通信网络由 4 个主要参数来描述。

(1) L (Latency): 表示源处理器与目的处理器进行消息 (一个或几个字) 通信所需的等待或延迟时间的上限，表示网络中消息的延迟。

(2) o (overhead): 表示处理器准备发送或接收每条消息的时间开销 (包括操作系统核心开销和网络软件开销)，在这段时间内处理器不能执行其他操作。

(3) g (gap): 表示一台处理器连续两次发送或接收消息时的最小时间间隔，其倒数即微处理器的通信带宽。

(4) P (Processor): 处理器/存储器模块个数。

1) LogP 模型的特点

(1) 抓住了网络与处理器之间的性能瓶颈。 g 反映了通信带宽, 单位时间内最多有 L/g 个消息能进行处理器间传送。

(2) 处理器间异步工作, 并通过处理器间的消息传送来完成同步。

(3) 对多线程技术有一定的反映。每台物理处理器可以模拟多台虚拟处理器 (VP), 当某台 VP 有访问请求时, 计算不会终止, 但 VP 的数目受限于通信带宽和上下文交换的开销。VP 受限于网络容量, 最多有 L/g 台 VP。

(4) 消息延迟不确定, 但延迟不大于 L 。消息经历的等待时间是不可预测的, 但在没有阻塞的情况下最大不超过 L 。

(5) LogP 模型鼓励编程人员采用一些好的策略, 如作业分配、计算与通信重叠及平衡的通信模式等。

(6) 可以预估算法的实际运行时间。

2) LogP 模型的不足

(1) 对网络中的通信模式描述得不够深入, 如对重发消息可能占满带宽、中间路由器缓存饱和等未加描述。

(2) LogP 模型主要适用于消息传递算法设计, 对于共享存储模式, 则简单地认为异地读操作相当于两次消息传递, 未考虑流水线预取技术、Cache 引起的数据不一致性及 Cache 命中率对计算的影响。

(3) 未考虑多线程技术的上下文开销。

(4) LogP 模型假设用点对点消息路由器进行通信, 这增加了编程者考虑路由器上相关通信操作的负担。

3. C3 模型

C3 模型假定处理器不能同时发送和接收消息, 它对超步的性能分析分为两部分: 计算单元 (CU), 依赖于本地计算量; 通信单元 (COU), 依赖于处理器发送和接收数据的多少、消息的延迟及通信引起的拥挤量。该模型考虑了两种路由 (存储转发路由和虫蚀寻径路由) 和两种发送/接收原语 (阻塞和无阻塞) 对 COU 的影响。

1) C3 模型的特点

(1) 用 C_l 和 C_p 来度量网络的拥挤对算法性能的影响。

(2) 考虑了不同路由和不同发送或接收原语对通信的影响。

(3) 不需要用户指定调度细节, 就可以评估超步的时间复杂性。

(4) 类似于 H-PRAM 模型的层次结构, C3 模型给编程者提供了 K 级路由算法的思路, 即系统被分为 K 级子系统, 各级子系统的操作相互独立, 用超步代替了 H-PRAM 中的 Sub PRAM 进行分割。

2) C3 模型的不足

- (1) C1 度量的前提为同一通信对中的两台处理器要分别位于对分网络的不同子网络内。
- (2) 该模型假设网络带宽等于处理器带宽，从而影响了正确描述可扩展系统。
- (3) 在 K 级算法中，处理器间的顺序可以有多种排列，但 C3 模型不能区分不同排列的难易程度。

4. BDM 模型

1996 年, J.F.JaJa 等人提出了一种块分布存储模型 (Block Distributed Model, BDM), 它是共享存储编程模式与基于消息传递的分布存储系统之间的桥梁模型。其主要有 4 个参数。

(1) P: 处理器个数。

(2) τ : 处理器从发出访问请求到得到远程数据的最大延迟时间, 包括准备请求时间、请求包在网络中路由的时间、目的处理器接收请求的时间, 以及将包中 M 个连续字返回给原处理器的时间。

(3) M: 局部存储器中连续的 M 个字。

(4) σ : 处理器发送数据到网络或从网络接收数据的时间。

1) BDM 模型的特点

(1) 用 M 反映出空间局部性特点, 提供了一种评价共享主存算法的性能方法, 度量了因远程访问引起的处理器间的通信。

(2) BDM 认可流水线技术。某台处理器的 K 次预取所需的时间为 $\tau + KM\sigma$ (否则为 $K(\tau + M\sigma)$), 可编程性好。

(3) 考虑了共享主存中的存储竞争问题。

(4) 可以用来分析网络路由情况。

2) BDM 模型的不足

(1) 认为初始数据置于内部存储中, 对于共享主存程序的编程者来说, 需要额外增加数据移动操作。

(2) 未考虑网络中影响延迟的因素, 如处理器的本地性、网络重拥挤等。

(3) 未考虑系统开销。

7.5.3 BSP 模型基本原理

BSP 模型是一种异步 MIMD-DM 模型 (DM——Distributed Memory, SM——Shared Memory), 其支持消息传递系统、块内异步并行、块间显式同步。该模型基于一个 Master 协调, 所有的 Worker 同步 (lock-step) 执行, 数据从输入的队列中读取。

BSP 计算模型不仅是一种体系结构模型, 也是设计并行程序的一种方法。BSP 程序的设计准则是整体同步 (Bulk Synchrony), 其独特之处在于超步 (Super Step) 概念的引入。一个 BSP 程序同时具有水平和垂直两个方向的结构。从垂直上看, 一个 BSP 程序由一系列串行的超步 (Super Step) 组

成,如图 7.28 所示,这种结构类似于一个串行程序结构。从水平上看,在一个超步中,所有的进程并行执行局部计算。一个超步可分为三个阶段,如图 7.29 所示。

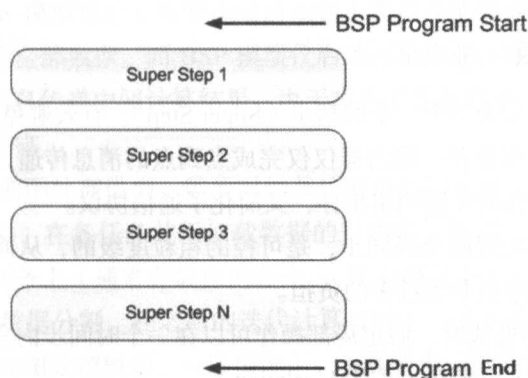


图 7.28

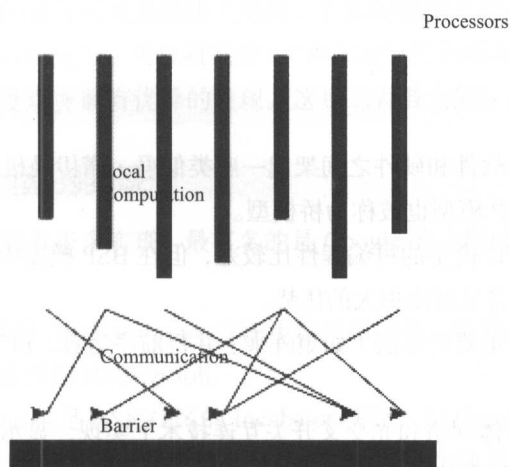


图 7.29

- 本地计算阶段,每台处理器只对存储在本地内存中的数据进行本地计算。
- 全局通信阶段,对任何非本地数据进行操作。
- 栅栏同步阶段,等待所有通信行为结束。

另外, BSP 并行计算模型可以用 p 、 s 、 g 、 i 4 个参数进行描述。

- p 为处理器的数目(带有存储器)。
- s 为处理器的计算速度。
- g 为每秒本地计算操作的数目/通信网络每秒传送的字节数,称之为选路器吞吐率,视为带宽因子。
- i 为全局的同步时间开销,称之为全局同步之间的时间间隔(Barrier Synchronization Time)。

假设有 p 台处理器同时传送 h 字节信息，则 gh 就是通信的开销。同步和通信的开销都规格化为处理器的指定条数。

7.5.4 BSP 模型的特点

- (1) BSP 模型将计算划分为一个一个的超步 (Super Step)，有效避免了死锁。
- (2) 它将处理器和路由器分开，路由器仅仅完成点到点的消息传递，不提供组合、复制和广播等功能，这样做既掩盖了具体的互连网络拓扑，又简化了通信协议。
- (3) 障碍同步是以硬件实现的全局同步，是可控的粗粒度级的，从而提供了执行紧耦合同步式并行算法的有效方式，而程序员并无过多的负担。
- (4) 在分析 BSP 模型的性能时，假定局部操作可以在一个时间步内完成，而在每一个超步中，一台处理器最多发送或接收 h 条消息 (称为 h -relation)。
- (5) 为 PRAM 模型设计的算法都可以采用在每台 BSP 处理器上模拟一些 PRAM 处理器的方法来实现。

7.5.5 BSP 模型的评价

- (1) 在并行计算时，Valiant 试图为软件和硬件之间架起一座类似冯·诺伊曼机的桥梁，BSP 模型可以起到这样的作用。正因如此，BSP 模型也被称为桥模型。
- (2) 一般而言，分布式存储的 MIMD 模型的可编程性比较差，但在 BSP 模型中，如果计算和通信可以适当平衡，则它在可编程性方面将呈现出更大的优势。
- (3) 在 BSP 模型中直接实现了一些重要的算法 (如矩阵乘、并行前序运算、FFT 和排序等)，均避免了自动存储管理的额外开销。
- (4) BSP 模型可以有效地在超立方体网络和光交叉开关互连技术上实现，显示出该模型与特定的技术实现无关，只需路由器具有一定的通信吞吐率。
- (5) 在 BSP 模型中，“超步”的长度必须能够充分地适应任意的 h -relation。
- (6) 在 BSP 模型中，在“超步”开始发送的消息，即使网络延迟时间比“超步”的长度短，该消息也只能在下一个“超步”中使用。
- (7) BSP 模型中的全局障碍同步假定是用特殊的硬件支持的，很多并行机中可能并没有相应的硬件。

7.5.6 BSP 与 MapReduce 对比

执行机制：MapReduce 是一个数据流模型，每个任务只对输入数据进行处理，产生的输出数据作为另一个任务的输入数据，并行任务之间独立地进行，串行任务之间以磁盘和数据复制作为交换介质和接口。而 BSP 是一个状态模型，各个子任务在本地的子图数据上执行计算、通信、修改图的

状态等操作，并行任务之间通过消息通信交流中间计算结果，不需要像 MapReduce 那样对全体数据进行复制。

迭代处理：MapReduce 模型理论上需要连续启动若干作业才能完成图的迭代处理，相邻作业之间通过分布式文件系统交换全部数据。而 BSP 模型仅启动一个作业，利用多个超步就可以完成迭代处理，两次迭代之间通过消息传递中间计算结果。由于减少了作业启动、调度开销和磁盘存取开销，BSP 模型的迭代执行效率较高。

数据分割：基于 BSP 的图处理模型，需要对加载后的图数据进行一次再分布的过程，以确定消息通信时的路由地址。例如，在各任务并行加载数据的过程中，根据一定的映射策略，将读入的数据重新分发到对应的计算任务上（通常存放在内存中），既有磁盘 I/O 又有网络通信，开销很大。但是一个 BSP 作业仅需一次数据分割，在之后的迭代计算过程中，除了消息通信外，无须进行数据的迁移。而基于 MapReduce 的图处理模型，一般情况下，不需要专门的数据分割处理。但是 Map 阶段和 Reduce 阶段存在中间结果的 Shuffle 过程，增加了磁盘 I/O 和网络通信开销。

MapReduce 的设计初衷是解决大规模、非实时数据处理问题。“大规模”决定了数据有局部特性可利用，从而可以划分、可以批处理；“非实时”代表响应时间较长，有充分的时间执行程序。而 BSP 模型在实时处理方面有优异的表现。这是二者最大的区别。

7.5.7 BSP 模型的实现

BSP 计算框架有很多实现，最有名的是 Google 的大规模图计算框架 Pregel，首次提出将 BSP 模型应用于图计算。

Yahoo! 贡献的 Apache Giraph 专注于迭代图计算（如 PageRank、最短连接等），每个 Job 就是一个没有 Reducer 过程的 Hadoop Job。

Apache Hama 也是 ASF 社区的 Incubator 项目，与 Giraph 不同的是，它是一个纯粹的 BSP 模型的 Java 实现，并且不仅用于图计算，而且意在提供一个通用的 BSP 模型的应用框架。下面将讨论一下开源的 Hama，让读者对 BSP 模型有更多的了解。

7.5.8 Apache Hama 简介

1. Hama 概述

2008 年 5 月，Hama 被视为 Apache 众多项目中一个被孵化的项目，是 Hadoop 项目的一个子项目。BSP 模型是 Hama 计算的核心，并且实现了分布式的计算框架，该框架可以用于矩阵计算（Matrix）、面向图计算（Graph）和网络计算（Network）。

Hama 是建立在 Hadoop 之上的分布式并行计算模型，基于 Map/Reduce 和 Bulk Synchronous 的实现框架，运行环境需要关联 ZooKeeper、HBase、HDFS 组件。集群环境中的系统架构由 BSPMaster/GroomServer（Computation Engine）、ZooKeeper（Distributed Locking）、HDFS/HBase（Storage Systems）

三大块组成。Hama 中有两个主要的模型：矩阵计算 (Matrix Package) 和面向图计算 (Graph Package)。

Hama 的主要应用领域包括：矩阵计算、面向图计算、PageRank、排序计算、BFS。

2. Hama 系统架构

Hama 主要由三部分组成：BSPMaster、GroomServer 和 ZooKeeper，HDFS 负责持久化存储数据 (如 job.jar)，BSPMaster 负责对 GroomServer 进行任务调配，GroomServer 负责分配和调度程序给 BSPPeer，BSPPeer 负责运行。

1) BSPMaster

在 Hama 中，BSPMaster 模块是一个主要角色，它主要负责协同各个计算节点之间的工作，每个计算节点在其注册到 Master 上的时候会分配到一个唯一的 ID。Master 内部维护着一张计算节点列表，表明当前哪些计算节点处于 Alive 状态。该列表中包括每个计算节点的 ID 和地址信息，以及哪些计算节点上被分配到了整个计算任务的哪一部分。Master 中这些信息的数据结构大小取决于整个计算任务被分成多少个 Partition。BSPMaster 负载并不高，因此普通配置即可。

下面来看看 BSPMaster 做了哪些工作：

- 维护 GroomServer 的状态。
- 控制在集群环境中的超步。
- 维护在 Groom 中 Job 的工作状态信息。
- 分配任务、调度任务到所有的 GroomServer 节点。
- 广播所有的 GroomServer 执行。
- 管理系统节点中的失效转发。
- 提供用户对集群环境的管理界面。

一个 BSPMaster 或者多台 GroomServer 是通过脚本启动的，在 GroomServer 中还包含了 BSPPeer 的实例，在启动 GroomServer 的时候就启动了 BSPPeer。BSPPeer 是整合在 GroomServer 中的，GroomServer 通过 PRC 代理与 BSPMaster 连接。当 BSPMaster、GroomServer 启动完毕以后，每个 GroomServer 的生命周期通过发送“心跳”信息给 BSPMaster 服务器，在这个“心跳”信息中包含了 GroomServer 的状态，这些状态包含了能够处理任务的最大容量和可用的系统内存状态等。

BSPMaster 的绝大部分工作，如 input、output、computation、saving 及 resuming from checkpoint，都会在一个叫作 Barrier 的地方终止。Master 会在每次操作中发送相同的指令到所有的计算节点，然后等待从每个计算节点的回应 (Response)。每次 BSP 主机接收到“心跳”消息以后，这个消息会带来最新的 GroomServer 状态，BSPMaster 根据 GroomServer 的可用资源对其进行任务调度和分配。BSPMaster 与 GroomServer 之间通信使用非常简单的 FIFO (先进先出) 原则。

2) GroomServer

GroomServer 由 BSPMaster 分配任务、接收任务，并且向 BSPMaster 处理报告状态，集群状态下的 GroomServer 需要运行在 HDFS 分布式存储环境中。而且对于 GroomServer 来说，一台 GroomServer 对应一个 BSPPeer 节点，需要运行在同一个物理节点上。

3) ZooKeeper

在 Hama 项目中, ZooKeeper 用来有效地管理 BSPPeer 节点之间的同步间隔 (Barrier Synchronization), 同时在系统失效转发的功能上发挥了重要的作用。

4) Hama 运行过程

Hama 运行过程如图 7.30 所示。

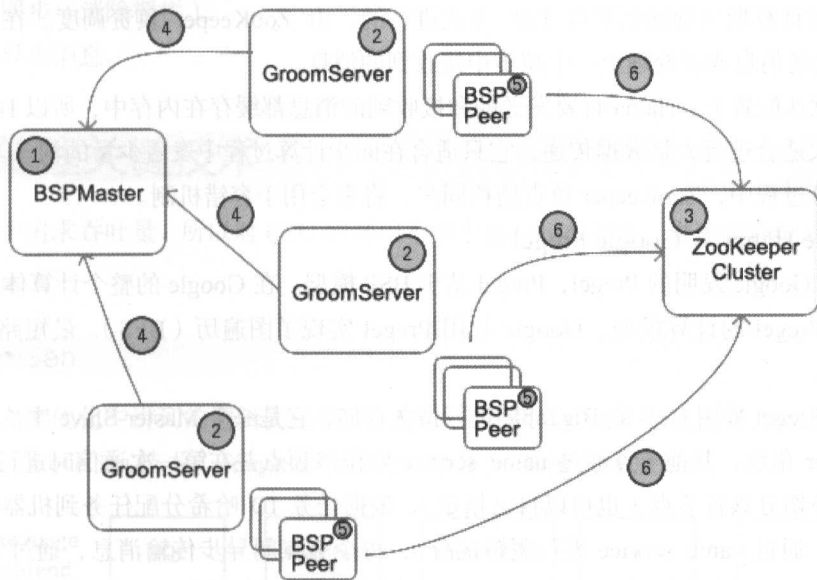


图 7.30

在图 7.30 中,

- ① BSPMaster 启动
- ② GroomServer 启动
- ③ ZooKeeper 集群启动
- ④ GroomServer 动态注册给 BSPMaster
- ⑤ GroomServer 初始化和管理工作 BSPPeer(s)
- ⑥ BSPPeer 通过 ZooKeeper 通信和栅栏同步

5) Apache Hama 作业流程

一个新的 Job 被提交后, BSPJobClient 先做一些初始化 Job 的工作, 如准备好作业的输入资源、代码等。

BSPMaster 将 Job 划分为一个个的 Task, 将 Task 分配给 GroomServer 去执行, 执行过程中维护 GroomServer 的进度与状态。GroomServer 发送心跳给 BSPMaster 来保持通信。超步的控制是由 BSPMaster 完成的。

GroomServer 启动 BSPPeer, 由 BSPPeer 来具体执行 Task。GroomServer 的主要任务是控制 BSPPeer

的启动和停止、维护任务的执行状态、向 BSPMaster 报告状态。一个 GroomServer 可以运行多个 Task，类似于 MapReduce 的 TaskTracker 的任务槽。所有 Task 都有一个 MasterTask，MasterTask 在整个计算开始和结束时分别调用 setup()和 cleanup()方法。如果该 GroomServer 下的一个 Task 失败，则 GroomServer 会重新启动这个 Task；如果 3 次重启 Task 都失败，则 GroomServer 向 BSPMaster 汇报该任务失败。

BSPPeer 在计算期间的通信是以 P2P 方式进行的，由 ZooKeeper 负责调度。在一个超步中，BSPPeer 只能发送消息或者处理上一个超步中接收到的消息。

另外，在默认配置下，Hama 将要发送的和接收到的消息都缓存在内存中，所以 Hama 本身的同步通信功能不太适合进行大量数据传递，它只适合在同步计算过程中发送少量的消息。

在整个计算过程中，ZooKeeper 负责栅栏同步，将来会用于容错机制。

6) Apache Hama 与 Google Pregel

Hama 类似 Google 发明的 Pregel，Pregel 基于 BSP 模型。在 Google 的整个计算体系中，有 20% 的计算依赖于 Pregel 的计算模型，Google 利用 Pregel 实现了图遍历（BFS）、最短路径（SSSP）、PageRank 计算。

Google 的 Pregel 采用 GFS 或 BigTable 进行持久存储，它是一个 Master-Slave 主从结构，有一个节点扮演 Master 角色，其他节点通过 name service 定位该顶点并在第一次通信时进行注册，Master 负责将计算任务切分到各节点（也可以自己指定），根据任务 ID 哈希分配任务到机器（一个机器可以有多个节点，通过 name service 进行逻辑区分），每个节点间异步传输消息，通过 Checkpoint 机制实现容错（更高级的容错通过 confined recovery 实现），并且每个节点向 Master 汇报心跳（ping），以便维持状态。

7) Hama 与 MapReduce 对比

① MapReduce 的不足

(1) MapReduce 主要针对松耦合型的数据处理应用，对于不容易分解成众多相互独立子任务的紧耦合型计算任务，处理效率很低。

(2) MapReduce 不能显式地支持迭代计算。

(3) MapReduce 是一种离线计算框架，不适合进行流式计算和实时分析。

② Hama 的优势

(1) 在科学计算领域的适用性：Hama 提供的基础组件能够适应多种需要矩阵和图形计算的应用。MapReduce 在单纯的大规模科学计算方面存在不足。比如，求一个大型矩阵的逆矩阵，需要进行大量的迭代计算，而读/写文件的次数并不多，此时 Hama 的迭代速度快的优势便体现出来了。

(2) 兼容性：Hama 能利用 Hadoop 及其相关的所有功能，因为 Hama 很好地兼容了现有的 Hadoop 接口。

(3) 可扩展性：得益于 Hama 的兼容性，Hama 能够充分利用大规模分布式接口的基础功能和服务，如亚马逊 EC2 无须任何修正就可以使用 Hama。

(4) 编程方式的灵活性: 为了保证灵活性, Hama 提供了简单计算引擎接口, 任何遵循此接口的计算引擎都能自由接入和退出。

③ Hama 亟待解决的问题

- (1) 完善容错能力。
- (2) NoSQL 的输入/输出格式。
- (3) 无视同步 (消除栅栏)。
- (4) 使用异步消息。

7.6 批处理关键技术

在批处理中追求吞吐量, 所以对 CPU 的利用率要求很高, 本节专门讲述两种批处理中提高 CPU 利用率的技术。

7.6.1 CodeGen^①

除了前面查询优化中讲到的逻辑优化器外, Spark 1.5 版本中更新较大的是 DataFrame 执行后端的优化, 引入了 CodeGen 技术 (Tungsten 项目的一部分), 如图 7.31 所示。

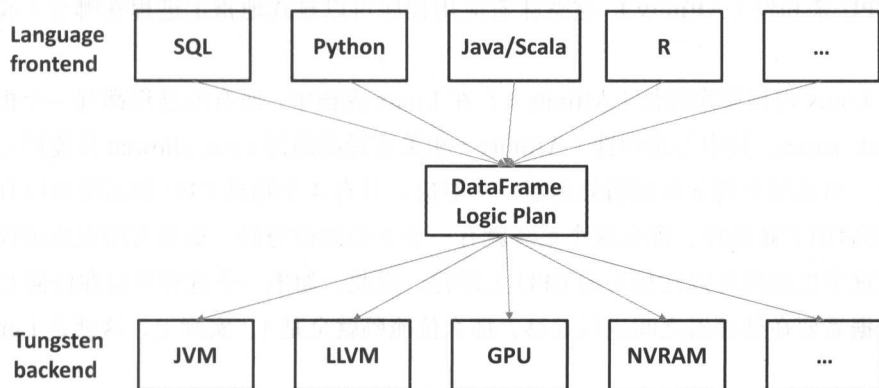


图 7.31

从图 7.31 中可以看出, Spark 通过 CodeGen 在运行前将逻辑计划生成对应的机器执行代码, 由 Tungsten backend 执行。

以 Spark 为代表的基于内存的计算引擎使得 I/O 性能比传统的基于硬盘的计算引擎有 10 倍左右的提升, 但与此同时, CPU 的瓶颈会更明显。以传统 PostgreSQL 的引擎为例, 操作数据都被缓存到内存的 Page Cache 上面, 执行最简单的 Count(*)统计只能勉强达到每秒 400 万行左右, 而真正需要

① 参考 <http://www.chinastor.com/a/spark/01201b462016.html>。

的操作其实是很少的。传统的数据库处理引擎有四大短板：其一是条件逻辑冗余，数据处理引擎代码非常烦琐；其二是虚函数的调用；其三是需要不断地从内存中调用数据，而无法一次性将数据从内存加载至 Cache；其四是为了保证数据引擎能跨不同的硬件平台，数据引擎很少支持一些扩展的指令集，这就导致本来可以提升的性能没有得到支持。

为了解决上述瓶颈，Google 研发的 Tenzing 技术里面提出基于 LLVM 编译框架实现动态生成代码的 CodeGen 技术，并且通过这个技术，基于 MapReduce 分布式框架下的类 SQL 系统的性能也能接近商业收费并行数据库的水准。

使用 CodeGen 的好处有：其一是简化了条件分支；其二是内存加载，可以使用代码生成来替代数据加载，从而极大地减少了内存的读取，增加了 CPU Cache 的利用率；其三是内联虚函数的调用；其四是能利用最新的指令集。

7.6.2 CPU 亲和技术^①

简单地说，CPU 亲和性 (Affinity) 是指进程要在某个给定的 CPU 上尽量长时间地运行而不被迁移到其他处理器的倾向性。

Linux 内核进程调度器天生就具有被称为软 CPU 亲和性 (Affinity) 的特性，这就意味着进程通常不会在处理器之间频繁迁移。2.6 版本的 Linux 内核还包含一种机制，它让开发人员可以编程实现硬 CPU 亲和性 (Affinity)，这意味着应用程序可以显式地指定进程在哪台 (或哪些) 处理器上运行。

什么是 Linux 内核硬亲和性 (Affinity)？在 Linux 内核中，所有的进程都有一个相关的数据结构，称为 task_struct。其中与亲和性 (Affinity) 相关度最高的是 cpus_allowed 位掩码。这个位掩码由 n 位组成，与系统中的 n 台逻辑处理器一一对应。具有 4 个物理 CPU 的系统可以有 4 位。如果这些 CPU 都启用了超线程，那么这个系统就有一个 8 位的位掩码。如果为给定的进程设置了给定的位，那么这个进程就可以在相关的 CPU 上运行。因此，如果一个进程可以在任何 CPU 上运行，并且能够根据需要在处理器之间进行迁移，那么位掩码就全是 1。实际上，这就是 Linux 中进程的默认状态。

Linux 内核 API 提供了一些方法，如 sched_set_affinity() (用来修改位掩码) 和 sched_get_affinity() (用来查看当前的位掩码)，让用户可以修改位掩码或查看当前的位掩码。

注意，cpu_affinity 会被传递给子线程，因此应该适当地调用 sched_set_affinity。

为什么应该使用硬亲和性 (Affinity)？

(1) 充分利用 CPU Cache。

(2) 保障时间敏感的、决定性的进程的 CPU 利用。

^① IBM 管理 CPU 的亲和性，<http://www.ibm.com/developerworks/cn/linux/l-affinity.html>。

使用 CPU 亲和技术会显著提高 CPU 利用率，但同时也会丧失程序的扩展性，因而应用程序需要单独设置。这项技术在一些需要高性能、软硬结合的场景下非常有效。

7.7 小结

批处理和流处理的场景不同，对平台的要求也不同。随着技术的发展，以及在客户对统一维护和降低学习成本的期望推动下，批处理和流处理技术逐渐统一。工业界也对批处理和流处理的融合进行了大量的尝试，如 Google 的 Dataflow、德国的 Flink 平台，以及 Spark 从 2.0 版本开始发力 Spark Streaming，等等。相信在不久的将来，就会实现二者的统一。

第 8 章

机器学习和数据挖掘

机器学习 (Machine Learning, ML) 是一门多领域交叉学科, 涉及概率论、统计学、逼近论、凸分析、算法复杂度理论等多门学科。其专门研究计算机是怎样模拟或实现人类的学习行为, 以获取新的知识或技能, 重新组织已有的知识结构, 使之不断改善自身的性能。

除了机器学习外, 本章还将涉及另外一个领域——数据挖掘, 它和机器学习有很大的交集。

机器学习和数据挖掘是两个非常难的领域, 本章更多地从架构和应用角度去解读, 理论知识则不进行重点阐述。

8.1 机器学习和数据挖掘的联系与区别

数据挖掘是从海量数据中获取有效的、新颖的、潜在有用的、最终可理解的模式的非平凡过程。数据挖掘中用到了大量的机器学习界提供的数据分析技术和数据库界提供的数据库管理技术。从数据分析的角度来看, 数据挖掘与机器学习有很多相似之处, 但不同之处也十分明显, 例如, 数据挖掘并没有机器学习探索人的学习机制这一科学发现任务, 数据挖掘中的数据分析是针对海量数据进行的, 等等。从某种意义上说, 机器学习的科学成分更重一些, 而数据挖掘的技术成分更重一些。

学习能力是智能行为的一个非常重要的特征, 不具有学习能力的系统很难称之为一个真正的智能系统, 而机器学习则希望 (计算机) 系统能够利用经验来改善自身的性能, 因此该领域一直是人工智能的核心研究领域之一。在计算机系统中, “经验” 通常是以数据的形式存在的, 因此, 机器学习不仅涉及对人的认知学习过程的探索, 还涉及对数据的分析处理。实际上, 机器学习已经成为计算机数据分析技术的创新源头之一。由于几乎所有的学科都要面对数据分析任务, 因此机器学习已经开始影响到计算机科学的众多领域, 甚至影响到计算机科学之外的很多学科。机器学习是数据挖掘中的一种重要工具。然而数据挖掘不仅仅要研究、拓展、应用一些机器学习方法, 还要通过许多非机器学习技术解决数据仓储、大规模数据、数据噪声等实践问题。机器学习的涉及面也很宽, 常用在数据挖掘上的方法通常只是 “从数据学习”。然而机器学习不仅仅可以用在数据挖掘上, 一些机器学习的子领域甚至与数据挖掘关系不大, 如增强学习与自动控制等。所以笔者认为, 数据挖掘是从目的而言的, 机器学习是从方法而言的, 两个领域有相当大的交集, 但不能等同。

8.2 典型的数据挖掘和机器学习过程

图 8.1 是一个典型的推荐类应用，需要找到“符合条件的”潜在人员。要从用户数据中得出这张列表，首先需要挖掘出客户特征，然后选择一个合适的模型来进行预测，最后从用户数据中得出结果。

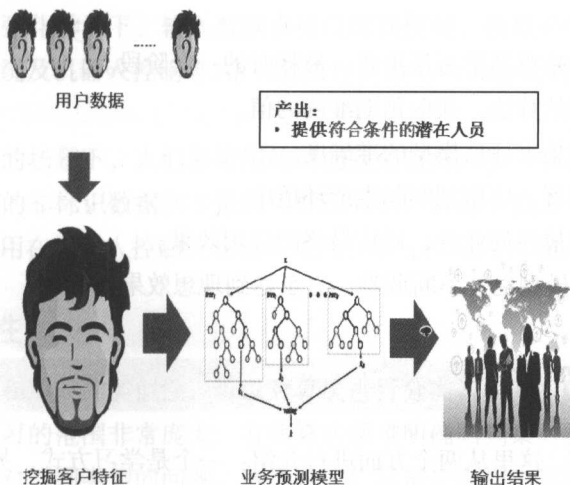


图 8.1

把上述例子中的用户列表获取过程进行细分，有如下几个部分（见图 8.2）。

act 典型数据挖掘

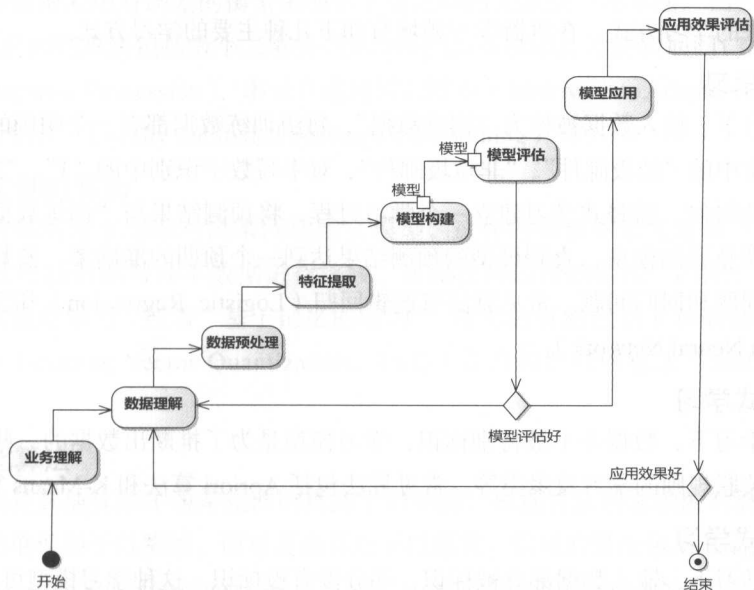


图 8.2

- 业务理解：理解业务本身，其本质是什么？是分类问题还是回归问题？数据怎么获取？应用哪些模型才能解决？
 - 数据理解：获取数据之后，分析数据里面有什么内容、数据是否准确，为下一步的预处理做准备。
 - 数据预处理：原始数据会有噪声，格式化也不好，所以为了保证预测的准确性，需要进行数据的预处理。
 - 特征提取：特征提取是机器学习最重要、最耗时的一个阶段。
 - 模型构建：使用适当的算法，获取预期准确的值。
 - 模型评估：根据测试集来评估模型的准确度。
 - 模型应用：将模型部署、应用到实际生产环境中。
 - 应用效果评估：根据最终的业务，评估最终的应用效果。
- 整个过程会不断反复，模型也会不断调整，直至达到理想效果。

8.3 机器学习概览

机器学习的算法有很多，这里从两个方面进行介绍：一个是学习方式，另一个是算法类似性。

8.3.1 学习方式

根据数据类型的不同，对一个问题的建模可以有不同的方式。在机器学习或人工智能领域，人们首先会考虑算法的学习方式。在机器学习领域有如下几种主要的学习方式。

1. 监督式学习

在监督式学习下，输入数据被称为“训练数据”，每组训练数据都有一个明确的标识或结果，如防垃圾邮件系统中的“垃圾邮件”、“非垃圾邮件”，对手写数字识别中的“1”、“2”、“3”、“4”等。在建立预测模型的时候，监督式学习建立一个学习过程，将预测结果与“训练数据”的实际结果进行比较，不断地调整预测模型，直到模型的预测结果达到一个预期的准确率。监督式学习的常见应用场景包括分类问题和回归问题。常见算法有逻辑回归（Logistic Regression）和反向传递神经网络（Back Propagation Neural Network）。

2. 非监督式学习

在非监督式学习下，数据并不被特别标识，学习模型是为了推断出数据的一些内在结构。常见的应用场景包括关联规则的学习及聚类等。常见算法包括 Apriori 算法和 K-Means 算法。

3. 半监督式学习

在半监督式学习下，输入数据部分被标识，部分没有被标识。这种学习模型可以用来进行预测，但是模型首先需要学习数据的内在结构，以便合理地组织数据进行预测。其应用场景包括分类和回

归。常见算法包括一些对常用监督式学习算法的延伸。这些算法首先试图对未标识的数据进行建模，然后在此基础上对标识的数据进行预测，如图论推理算法（Graph Inference）或拉普拉斯支持向量机（Laplacian SVM）等。

4. 强化学习

在强化学习下，输入数据作为对模型的反馈，不像监督模型那样，输入数据仅作为一种检查模型对错的方式。在强化学习下，输入数据直接反馈到模型，模型必须对此立刻做出调整。常见的应用场景包括动态系统及机器人控制等。常见算法包括 Q-Learning 及时间差学习（Temporal Difference Learning）等。

在企业数据应用的场景下，人们最常用的可能就是监督式学习和非监督式学习。在图像识别等领域，由于存在大量的非标识数据和少量的可标识数据，目前半监督式学习是一个很热门的话题。而强化学习更多地应用在机器人控制及其他需要进行系统控制的领域。

8.3.2 算法类似性

根据算法的功能和形式的类似性，可以对算法进行分类，如基于树的算法、基于神经网络的算法等。当然，机器学习的范围非常庞大，有些算法很难明确归到某一类。而对于有些分类来说，同一分类的算法可以针对不同类型的问题。这里，我们尽量把常用的算法按照最容易理解的方式进行分类。

1. 回归算法

回归算法是试图采用对误差的衡量来探索变量之间的关系的一类算法。回归算法是统计机器学习的利器。常见的回归算法包括最小二乘法（Ordinary Least Square）、逻辑回归（Logistic Regression）、逐步式回归（Stepwise Regression）、多元自适应回归样条（Multivariate Adaptive Regression Splines）及本地散点平滑估计（Locally Estimated Scatterplot Smoothing）等。

2. 基于实例的算法

基于实例的算法常常用来对决策问题建立模型，这样的模型常常先选取一批样本数据，然后根据某些近似性把新数据与样本数据进行比较，从而找到最佳的匹配。因此，基于实例的算法常常被称为“赢家通吃学习”或者“基于记忆的学习”。常见的算法包括 k-Nearest Neighbor（kNN）、学习矢量量化（Learning Vector Quantization, LVQ）及自组织映射算法（Self-Organizing Map, SOM）等。

3. 正则化算法

正则化算法是其他算法（通常是回归算法）的延伸，根据算法的复杂度对算法进行调整。正则化算法通常对简单模型予以奖励，而对复杂算法予以惩罚。常见的算法包括 Ridge Regression、Least Absolute Shrinkage and Selection Operator（LASSO）及弹性网络（Elastic Net）等。

4. 决策树算法

决策树算法根据数据的属性采用树状结构建立决策模型，常常用来解决分类和回归问题。常见的算法包括分类及回归树（Classification and Regression Tree, CART）、ID3（Iterative Dichotomiser 3）、C4.5、Chi-squared Automatic Interaction Detection（CHAID）、Decision Stump、随机森林（Random Forest）、多元自适应回归样条（MARS）及梯度推进机（Gradient Boosting Machine, GBM）等。

5. 贝叶斯算法

贝叶斯算法是基于贝叶斯定理的一类算法，主要用来解决分类和回归问题。常见的算法包括朴素贝叶斯算法、平均单依赖估计（Averaged One-Dependence Estimators, AODE）及 Bayesian Belief Network（BBN）等。

6. 基于核的算法

基于核的算法中最著名的莫过于支持向量机（SVM）。基于核的算法是把输入数据映射到一个高阶的向量空间，在这些高阶向量空间里，有些分类或者回归问题能够更容易地解决。常见的基于核的算法包括支持向量机（Support Vector Machine, SVM）、径向基函数（Radial Basis Function, RBF）及线性判别分析（Linear Discriminate Analysis, LDA）等。

7. 聚类算法

聚类算法通常按照中心点或者分层的方式对输入数据进行归并。所有的聚类算法都试图找到数据的内在结构，以便按照最大的共同点将数据进行归类。常见的聚类算法包括 K-Means 算法及期望最大化算法（Expectation Maximization, EM）等。

8. 关联规则学习

关联规则学习通过寻找最能够解释数据变量之间关系的规则，来找出大量多元数据集中有用的关联规则。常见的算法包括 Apriori 算法和 Eclat 算法等。

9. 人工神经网络算法

人工神经网络算法模拟生物神经网络，是一类模式匹配算法，通常用于解决分类和回归问题。人工神经网络是机器学习的一个庞大的分支，有几百种不同的算法（深度学习就是其中的一类算法）。常见的人工神经网络算法包括感知器神经网络（Perceptron Neural Network）、反向传递（Back Propagation）、Hopfield 网络、自组织映射（Self-Organizing Map, SOM）及学习矢量量化（Learning Vector Quantization, LVQ）等。

10. 深度学习算法

深度学习算法是对人工神经网络的发展。在计算能力变得日益廉价的今天，深度学习算法试图建立大得多也复杂得多的神经网络。很多深度学习算法是半监督式学习算法，用来处理存在少量未标识数据的大数据集。常见的深度学习算法包括受限波尔兹曼机（Restricted Boltzmann Machine, RBN）、Deep Belief Networks（DBN）、卷积网络（Convolutional Network）及堆栈式自动编码器（Stacked Auto-encoders）等。

11. 降低维度算法

与聚类算法一样,降低维度算法试图分析数据的内在结构,不过降低维度算法通过非监督式学习,试图利用较少的信息来归纳或者解释数据。这类算法可以用于高维数据的可视化,或者用来简化数据以便监督式学习使用。常见的降低维度算法包括主成分分析(Principle Component Analysis, PCA)、偏最小二乘回归(Partial Least Square Regression, PLSR)、Sammon 映射、多维尺度(Multi-Dimensional Scaling, MDS)及投影追踪(Projection Pursuit)等。

12. 集成算法

集成算法用一些相对较弱的学习模型独立地就同样的样本进行训练,然后把结果整合起来进行整体预测。集成算法的主要难点在于究竟集成哪些独立的、较弱的学习模型,以及如何把学习结果整合起来。这是一类非常强大的算法,同时也非常流行。常见的集成算法包括 Boosting、Bootstrapped Aggregation(Bagging)、AdaBoost、堆叠泛化(Stacked Generalization, Blending)、梯度推进机(Gradient Boosting Machine, GBM)及随机森林(Random Forest)等。

8.4 机器学习&数据挖掘应用案例

前面了解了机器学习和数据挖掘的基本概念,本节来看一下业界成熟的案例,让读者对机器学习和数据挖掘有一个直观的理解。

8.4.1 尿布和啤酒的故事

先来看一则有关数据挖掘的故事——“尿布与啤酒”。

总部位于美国阿肯色州的世界著名商业零售连锁企业沃尔玛拥有世界上最大的数据仓库系统。为了能够准确了解顾客在其门店的购买习惯,沃尔玛对其顾客的购物行为进行购物篮分析,想知道顾客经常一起购买的商品有哪些。沃尔玛数据仓库里集中了其各门店的详细原始交易数据,在这些原始交易数据的基础上,沃尔玛利用 NCR 数据挖掘工具对这些数据进行分析 and 挖掘。一个意外的发现是:跟尿布一起购买最多的商品竟然是啤酒!这是数据挖掘技术对历史数据进行分析的结果,反映了数据的内在规律。那么,这个结果符合现实情况吗?是否有利用价值?

于是,沃尔玛派出市场调查人员和分析师对这一数据挖掘结果进行调查分析,从而揭示出隐藏在“尿布与啤酒”背后的美国人的一种行为模式:在美国,一些年轻的父亲下班后经常要到超市去买婴儿尿布,而他们有 30%~40%的人同时也为自己买一些啤酒。产生这一现象的原因是:美国的太太们常叮嘱她们的丈夫下班后为小孩买尿布,而丈夫们在买完尿布后又随手带回了他们喜欢的啤酒。

既然尿布与啤酒一起被购买的机会很多,于是沃尔玛就在其各家门店将尿布与啤酒摆放在一起,结果是尿布与啤酒的销售量双双增长。

8.4.2 决策树用于电信领域故障快速定位

电信领域比较常见的应用场景是决策树，利用决策树来进行故障定位。比如，用户投诉上网慢，其中就有很多种原因，有可能是网络的问题，也有可能是用户手机的问题，还有可能是用户自身感受的问题。怎样快速分析和定位出问题，给用户一个满意的答复？这就需要用到决策树。

图 8.3 就是一个典型的用户投诉上网慢的决策树的样例。

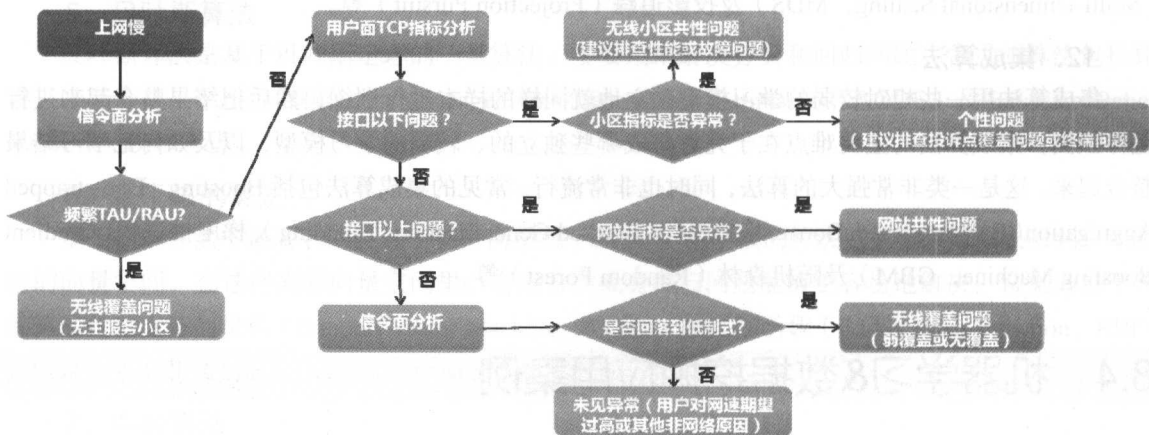


图 8.3

8.4.3 图像识别领域

1. 小米面孔相册

这项功能的名字叫“面孔相册”，可以利用图像分析技术，自动地对云相册照片内容按照面孔进行分类整理。开启“面孔相册”功能后，可以自动识别、整理和分类云相册中的不同面孔。

“面孔相册”还支持手动调整分组、移出错误面孔、通过系统推荐确认面孔等功能，从而弥补机器识别的不足。

这项功能的背后其实使用的是深度学习技术，自动识别图片中的人脸，然后进行自动识别和分类。

2. 支付宝扫脸支付

马云在 2015 CeBIT 展会开幕式上首次展示了蚂蚁金服的最新支付技术“Smile to Pay”（扫脸支付），惊艳全场。支付宝宣称，Face++ Financial 人脸识别技术在 LFW 国际公开测试集中达到 99.5% 的准确率，同时还能运用“交互式指令+连续性判定+3D 判定”技术。人脸识别技术基于神经网络，让计算机学习人的大脑，并通过“深度学习算法”大量训练，让它变得极为“聪明”，能够“认人”。实现人脸识别不需要用户自行提交照片，有资质的机构在需要进行人脸识别时，可以向全国公民身份证号码查询服务中心提出申请，将采集到的照片与该部门的权威照片库进行比对。

也就是说，用户在进行人脸识别时，只需打开手机或电脑的摄像头，对着自己的正脸进行拍摄

即可。在智能手机全面普及的今天,这个参与门槛低到可以忽略不计。

用户容易担心的隐私问题在人脸识别领域也能有效避免,因为照片来源权威,同时,一种特有的“脱敏”技术可以将照片模糊处理成肉眼无法识别而只有计算机才能识别的图像。

3. 图片内容识别

前面两个案例介绍的都是图片识别,比图片识别更难的是图片语义的理解和提取,百度和 Google 都在进行这方面的研究。

百度的百度识图能够有效地处理特定物体的检测识别(如人脸、文字或商品)、通用图像的分类标注,如图 8.4 所示。



图 8.4

来自 Google 研究院的科学家发表了一篇博文,展示了 Google 在图形识别领域的最新研究进展。或许未来 Google 的图形识别引擎不仅能够识别出图片中的对象,还能够对整个场景进行简短而准确的描述,如图 8.5 所示。这种突破性的概念来自机器语言翻译方面的研究成果:通过一种递归神经网络(RNN)将一种语言的语句转换成向量表达,并采用第二种 RNN 将向量表达转换成目标语言的语句。



图 8.5

而 Google 将以上过程中的第一种 RNN 用深度卷积神经网络 CNN 替代,这种网络可以用来识别图像中的物体。通过这种方法可以实现将图像中的对象转换成语句,对图像场景进行描述。概念虽

然简单，但实现起来十分复杂，科学家表示目前实验产生的语句合理性不错，但距离完美仍有差距，这项研究目前仅处于早期阶段。图 8.6 展示了通过此方法识别图像对象并产生描述的过程。

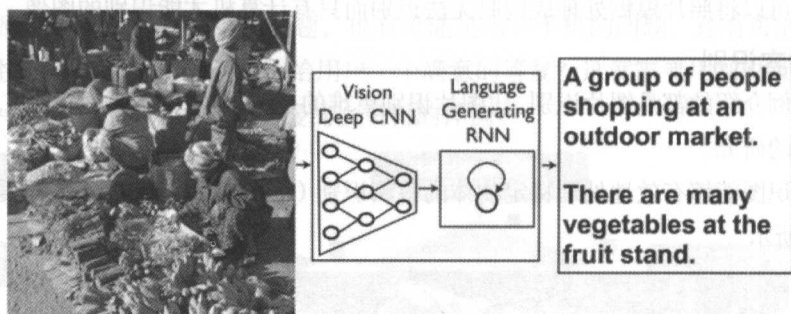


图 8.6

8.4.4 自然语言识别

自然语言识别一直是一个非常热门的领域，最有名的是苹果的 Siri，支持资源输入，调用手机自带的天气预报、日常安排、搜索资料等应用，还能够不断学习新的声音和语调，提供对话式的应答。

微软的 Skype Translator 可以实现中英文之间的实时语音翻译功能，将使得英文和中文普通话之间的实时语音对话成为现实。

Skype Translator 的运作机制如图 8.7 所示。



图 8.7

在准备好的数据被录入机器学习系统后，机器学习软件会在这些对话和环境涉及的单词中搭建一个统计模型。当用户说话时，软件会在该统计模型中寻找相似的内容，然后应用到预先“学到”的转换程序中，将音频转换为文本，再将文本转换成另一种语言。

虽然语音识别一直是近几十年来的重要研究课题，但是该技术的发展普遍受到错误率高、麦克风敏感度差异、噪声环境等因素的阻碍。将深层神经网络（DNNs）技术引入语音识别，极大地降低了错误率、提高了可靠性，最终使这项语音翻译技术得以广泛应用。

8.5 交互式分析^①

在机器学习和数据挖掘领域，有“数据科学家”这样一种职位。“数据科学家”在2009年由Natahn Yau首次提出，是指能采用科学方法、运用数据挖掘工具，对复杂多量的数字、符号、文字、网址、音频或视频等信息进行数字化重现与认识，并能寻找新的数据洞察的工程师或专家（不同于统计学家或分析师）。一个优秀的数据科学家需要具备的素质包括：懂数据采集、懂数学算法、懂数学软件、懂数据分析、懂预测分析、懂市场应用、懂决策分析等。

传统的典型应用（如推荐系统）的一个数据流过程，需要经历“使用Hadoop做ETL→使用Impala/Drill等做数据探索→使用Tableau做报表→使用R语言或者Mahout做高级分析→最终形成一个数据产品”等过程，如图8.8所示。

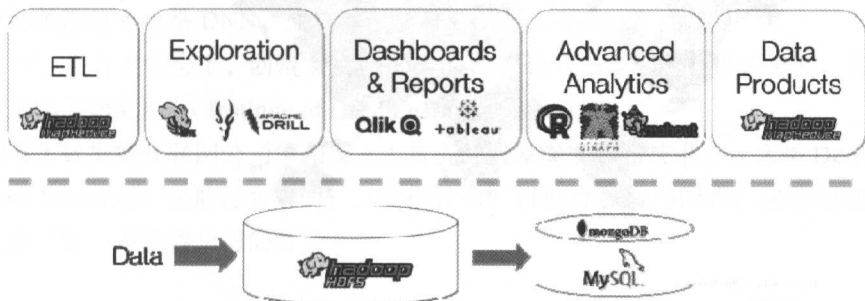


图 8.8

这个过程非常复杂，对技能要求非常高，需要懂一系列复杂的系统和工具。Databricks 创新地将 ETL、探索、高级分析、报表、数据产品统一到一个平台上，如图 8.9 所示。

这里用到的核心工具是 Notebooks。Notebooks 提供一个交互式的工作区，数据科学家可以使用 R、Python、Scala、SQL 等语言直接在工作区中进行输入，结果将以图形化的方式展现。移动设备的地理分布如图 8.10 所示。

① 参见 <https://databricks.com/blog/2014/07/14/databricks-cloud-making-big-data-easy.html>。

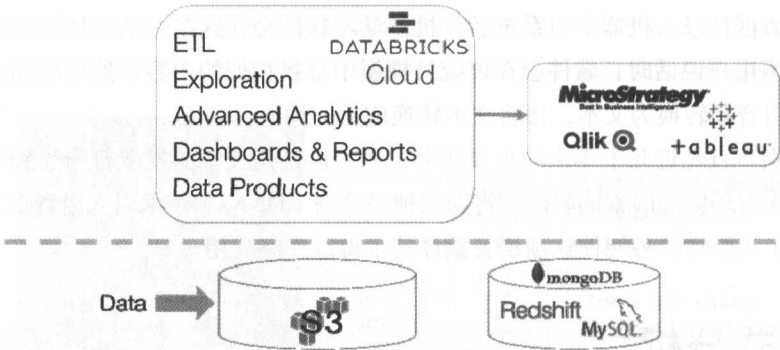


图 8.9

Mobile Devices by Geography (Sample Data)

This is a world map of number of mobile phones by country from a sample dataset



图 8.10

8.6 深度学习^{①②}

8.6.1 深度学习概述

Artificial Intelligence (人工智能) 是人类美好的愿望之一。虽然计算机技术已经取得了长足的进步，但截至目前，还没有一台计算机能够产生“自我”的意识。的确，在人类和大量现有数据的帮

① Deep Learning (深度学习) 学习笔记整理系列, <http://blog.csdn.net/zouxy09/article/details/8775360>。

② 一文读懂机器学习, <http://www.36dsj.com/archives/20382>。

助下,计算机可以表现得十分强大,但是离开了这两者,它甚至都不能分辨两只小动物。

1950年提出图灵试验(图灵是计算机和人工智能的鼻祖,分别对应于著名的“图灵机”和“图灵测试”)的设想,即隔墙对话,你将不知道与你对话的是人还是机器。这无疑给计算机,尤其是人工智能,预设了一个很高的期望值。但是半个世纪过去了,人工智能的进展远远没有达到图灵试验的标准。这不仅让多年翘首以待的人们心灰意冷,更让人们认为相关领域是“伪科学”。

自2006年以来,机器学习领域取得了突破性的进展,图灵试验至少不再那么可望而不可即。至于技术手段,不仅依赖于云计算对大数据的并行处理能力,而且依赖于算法。这个算法就是 Deep Learning。借助 Deep Learning 算法,人类终于找到了如何处理“抽象概念”这个亘古难题的方法。

2012年6月,《纽约时报》披露了 Google Brain 项目,吸引了公众的广泛关注。这个项目是由斯坦福大学的机器学习教授 Andrew Ng 和大规模计算机系统方面的世界顶尖专家 Jeff Dean 共同主导的,他们用 16 000 个 CPU Core 的并行计算平台训练一种称为“深度神经网络”(Deep Neural Networks, DNN)的机器学习模型(内部共有 10 亿个节点),在语音识别和图像识别等领域获得了巨大的成功。

项目负责人之一 Andrew 称:“我们没有像通常做的那样自己框定边界,而是直接把海量数据投放到算法中,让数据自己说话,系统会自动从数据中学习。”另外一名负责人 Jeff 则说:“我们在训练的时候从来不会告诉机器‘这是一只猫’,系统其实是自己发明或者领悟了‘猫’的概念。”

2012年11月,微软在中国天津的一次活动上公开演示了一个全自动的同声传译系统,讲演者用英文演讲,后台的计算机自动完成语音识别、中英机器翻译和中文语音合成,效果非常流畅。据报道,后面支撑的关键技术也是 DNN,或者深度学习(Deep Learning, DL)。

2013年1月,在百度年会上,创始人兼 CEO 李彦宏高调宣布要成立百度研究院,其中第一个成立的就是“深度学习研究所”(Institute of Deep Learning, IDL)。

为什么拥有大数据的互联网公司争相投入大量资源研发深度学习技术。什么是 Deep Learning? 为什么有 Deep Learning? 它是怎么来的? 又能做什么? 目前存在哪些困难? 这些问题需要慢慢解答。我们首先来了解一下机器学习的背景。

8.6.2 机器学习的背景

机器学习(Machine Learning)是一门专门研究计算机怎样模拟或实现人类的学习行为,以获取新的知识或技能,重新组织已有的知识结构,使之不断改善自身性能的学科。机器能否像人类一样具有学习能力呢? 1959年,美国的塞缪尔(Samuel)设计了一个下棋程序,这个程序具有学习能力,它可以在不断地对弈中改善自己的棋艺。这个程序向人们展示了机器学习的能力,提出了很多令人深思的社会问题与哲学问题。

机器学习虽然发展了几十年,但仍然存在很多没有得到良好解决的问题,如图像识别、语音识别、自然语言理解、天气预测、基因表达、内容推荐等。目前通过机器学习去解决这些问题的思路如图 8.11 所示(以视觉感知为例)。

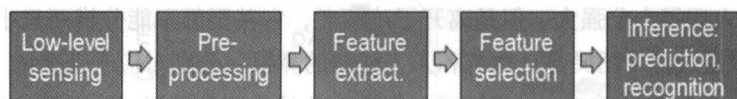


图 8.11

首先通过传感器（如 CMOS）获取数据；然后经过预处理、特征提取、特征选择；再到推理、预测或者识别；最后，也就是机器学习的部分，绝大部分工作是在这里做的，也存在很多论文和研究。

而中间的几部分（特征提取、特征选择，再到推理、预测或者识别）概括起来就是特征表达。良好的特征表达对最终算法的准确性起了非常关键的作用，而且系统主要的计算和测试工作都耗费在这一部分。但是这一部分实际上是由人工完成的，即依靠人工提取特征。

截至目前，出现了很多好的特征（好的特征应具有不变性和可区分性），如 SIFT 的出现是局部图像特征描述子研究领域一项里程碑式的工作。由于 SIFT 对尺度、旋转及一定视角和光照变化等图像变化都具有不变性，并且具有很强的可区分性，因此让很多问题的解决变为可能，但它也不是万能的。

然而，手工选取特征是一种非常费力、启发式（需要专业知识）的方法，能不能选取好很大程度上要靠经验和运气，而且它的调节需要大量的时间。既然手工选取特征性能不佳，那么，能不能自动学习一些特征呢？答案是肯定的。Deep Learning 就是设计用来做这件事情的。

那它是怎么学习的呢？怎么知道哪些特征好、哪些不好呢？我们说机器学习是一门专门研究计算机怎样模拟或实现人类的学习行为的学科，那么人类的视觉系统是怎样工作的呢？我们能不能参考人脑、模拟人脑呢？

近几十年来，认知神经科学、生物学等学科的发展，让我们对神秘而又神奇的大脑不再那么陌生，也为人工智能的发展推波助澜。

8.6.3 人脑视觉机理

1981 年的诺贝尔医学奖颁给了 David Hubel（出生于加拿大的美国神经生物学家）和 Torsten Wiesel，以及 Roger Sperry。前两位的主要贡献是“发现了视觉系统的信息处理”：可视皮层是分级的，如图 8.12 所示。

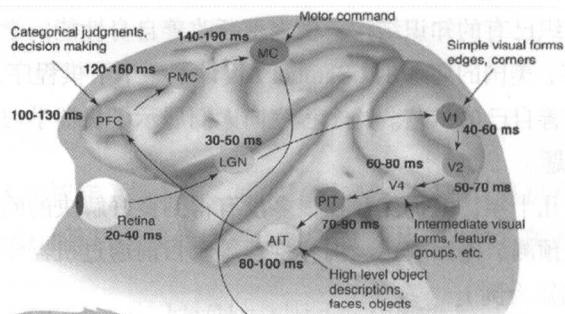


图 8.12

我们来看看他们都做了什么。1958 年, David Hubel 和 Torsten Wiesel 在 John Hopkins University 研究瞳孔区域与大脑皮层神经元的对应关系。他们首先在猫的后脑头骨上开了一个深 3 毫米的小洞, 向洞里插入电极, 测试神经元的活跃程度。

然后, 他们在猫的眼前展现各种形状、各种亮度的物体, 并且在展现每一件物体时, 同时改变物体放置的位置和角度。他们期望通过这种方法, 让猫的瞳孔感受不同类型、不同强弱的刺激。

之所以做这个试验, 目的是去证明一个猜测: 位于后脑皮层的不同视觉神经元与瞳孔所受刺激之间存在某种对应关系, 一旦瞳孔受到某种刺激, 后脑皮层的某一部分神经元就会活跃。经过反复试验, David Hubel 和 Torsten Wiesel 发现了一种被称为“方向选择性细胞 (Orientation Selective Cell)”的神经元细胞。当瞳孔发现了眼前的物体的边缘, 而且这个边缘指向某个方向时, 这种神经元细胞就会活跃。

这个发现激发了人们对于神经系统的进一步思考。神经—中枢—大脑的工作过程, 或许就是一个不断迭代、不断抽象的过程。

这里有两个关键词: 一个是抽象, 另一个是迭代。从原始信号做低级抽象, 逐渐向高级抽象迭代。人类的逻辑思维经常使用高度抽象的概念。

例如, 从原始信号摄入开始 (瞳孔摄入像素), 接着做初步处理 (大脑皮层某些细胞发现边缘和方向), 然后抽象 (大脑判定眼前物体的形状是圆形的), 最后进一步抽象 (大脑进一步判定该物体是一只气球), 如图 8.13 所示。

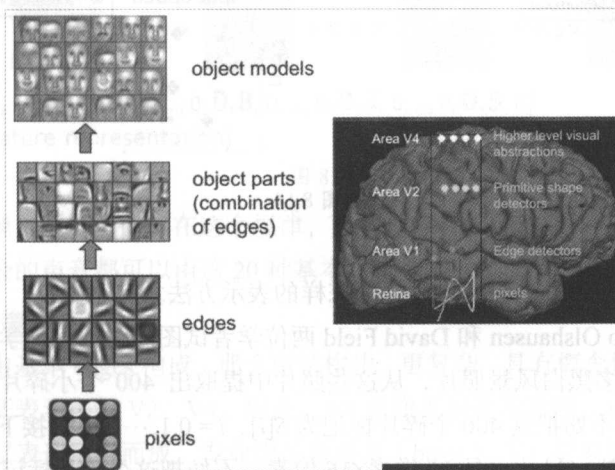


图 8.13

总的来说, 人类视觉系统的信息处理是分级的。从低级的 V1 区提取边缘特征, 再到 V2 区的形状或者目标等, 最后到更高层, 即整个目标、目标的行为等。也就是说高层特征是低层特征的组合, 从低层到高层的特征表示越来越抽象, 越来越能表现语义或者意图。而抽象层面越高, 存在的可能猜测就越少, 就越有利于分类。例如, 单词集合和句子的对应关系是多对一的, 句子和语义的对应关系又是多对一的, 语义和意图的对应关系还是多对一的, 这是一个层级体系。

敏感的人注意到一个关键词：分层。而 Deep Learning 的 Deep 是不是就表示存在很多层呢？答案是肯定的。那么，Deep Learning 是如何借鉴这个过程的呢？

因为我们要学习的是特征的表达，那么关于特征，或者说关于这个层级特征，我们需要了解得更深入一些。

8.6.4 关于特征

特征是机器学习系统的原材料，对最终模型的影响是毋庸置疑的。如果数据被很好地表达成了特征，那么线性模型通常就能达到满意的精度。对于特征，我们需要考虑哪些方面呢？

1. 特征表示的粒度

学习算法在一个什么粒度上的特征表示才能发挥作用？就一张图片来说，像素级的特征根本没有价值。例如图 8.14 所示的摩托车，从像素级别根本得不到任何信息，无法进行摩托车和非摩托车的区分。而如果特征具有结构性（或者说有含义）的时候，如是否具有车把手（Handle）、是否具有车轮（Wheel），就很容易进行区分，学习算法才能发挥作用。

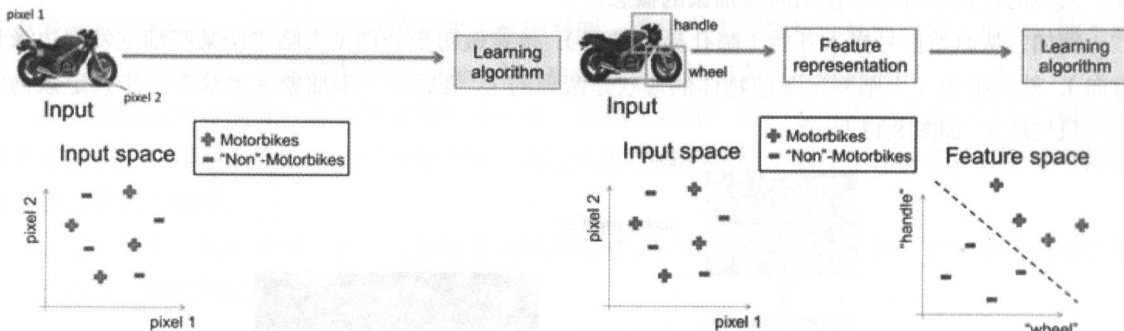


图 8.14

2. 初级（浅层）特征表示

既然像素级的特征表示方法没有作用，那怎样的表示方法才有用呢？

1995 年前后，Bruno Olshausen 和 David Field 两位学者试图同时用生理学和计算机的手段研究视觉问题。他们收集了很多黑白风景照片，从这些照片中提取出 400 个小碎片，每个照片碎片的尺寸均为 16 像素×16 像素。不妨把这 400 个碎片标记为 $S[i]$, $i = 0, 1, \dots, 399$ 。接下来，从这些黑白风景照片中随机提取另一个碎片，尺寸也是 16 像素×16 像素，不妨把这个碎片标记为 T 。

他们提出的问题是：如何从这 400 个碎片中选取一组碎片 $S[k]$ ，通过叠加的方法合成一个新的碎片，而这个新的碎片应当与随机选择的目标碎片 T 尽可能相似，同时 $S[k]$ 的数量尽可能少？用数学的语言来描述就是： $\sum_k (a[k] \times S[k]) \rightarrow T$ ，其中， $a[k]$ 是在叠加碎片 $S[k]$ 时的权重系数。

为了解决这个问题，Bruno Olshausen 和 David Field 发明了一种算法——稀疏编码（Sparse Coding）。

稀疏编码是一个重复迭代的过程，每次迭代分为两步：

(1) 选择一组 $S[k]$ ，然后调整 $a[k]$ ，使得 $\text{Sum}_k(a[k] \times S[k])$ 最接近 T 。

(2) 固定 $a[k]$ ，在 400 个碎片中，选择其他更合适的碎片 $S'[k]$ ，替代原先的 $S[k]$ ，使得 $\text{Sum}_k(a[k] \times S'[k])$ 最接近 T 。

经过几次迭代后，最佳的 $S[k]$ 组合被遴选出来。令人惊奇的是，被选中的 $S[k]$ 基本上都是照片上不同物体的边缘线，这些线段形状相似，区别在于方向。

Bruno Olshausen 和 David Field 的算法结果与 David Hubel 和 Torsten Wiesel 的生理发现不谋而合。

也就是说，复杂图形往往由一些基本结构组成。如图 8.15 所示，一张图可以通过用 64 种正交的 edges（可以理解为正交的基本结构）来线性表示。比如，样例 x 可以用 1~64 个 edges 中的 3 个按照 0.8、0.3、0.5 的权重调和而成，而其他 edges 没有贡献，因此均为 0。

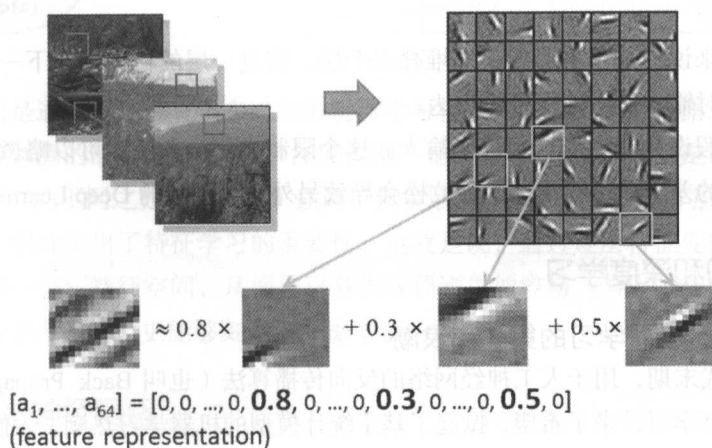


图 8.15

另外，学者还发现，不仅图像存在这个规律，声音也存在。他们从未标注的声音中发现了 20 种基本的声音结构，其余的声音都可以由这 20 种基本结构合成。

3. 结构性特征表示

小块的图形可以由基本 edges 构成，那么更结构化、更复杂、具有概念性的图形如何表示呢？这就需要更高层次的特征表示，如 V2、V4。因此，V1 看像素级是像素级，V2 看 V1 是像素级，逐层递进，高层表达由低层表达组合而成。专业说法就是基（basis）。V1 层提取出的 basis 是边缘，V2 层是 V1 层这些 basis 的组合，这时候 V2 层得到的又是高一层的 basis。

8.6.5 需要有多少个特征

任何一种方法，特征越多，给出的参考信息就越多，准确性就会得到提升。但特征多意味着计算复杂、探索空间大，可以用来训练的数据在每个特征上就会稀疏，因此会带来各种问题，所以说并不是特征越多越好。

上述讨论得出一个结论：Deep Learning 需要用多层来获得更抽象的特征表达。那么多少层才合适呢？用什么架构来建模呢？怎么进行非监督训练呢？

8.6.6 深度学习的基本思想

假设有一个系统 S ，它有 n 层 (S_1, S_2, \dots, S_n)，它的输入是 I ，输出是 O ，形象地表示为： $I \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \dots \Rightarrow S_n \Rightarrow O$ ，如果输出 O 等于输入 I ，则输入 I 经过这个系统变化后没有任何信息损失。设处理 a 信息得到 b ，再处理 b 得到 c ，那么可以证明： a 和 c 的互信息不会超过 a 和 b 的互信息。这表明信息处理不会增加信息，大部分处理会丢失信息。现在回到主题 Deep Learning，我们需要自动地学习特征。假设有一堆输入 I （一堆图像或者文本），设计了一个系统 S （有 n 层），通过调整系统中的参数，使得它的输出仍然是输入 I ，那么我们就可以自动获取得到输入 I 的一系列层次特征，即 S_1, S_2, \dots, S_n 。

对于深度学习来说，其基本思想就是堆叠多个层，将这一层的输出作为下一层的输入。通过这种方式就可以实现对输入信息进行分级表达。

另外，前面的假设是输出严格地等于输入。这个限制太严格，我们可以略微地放松这个限制，如使得输入与输出的差别尽可能小。这种放松会导致另外一种不同的 Deep Learning 方法。

8.6.7 浅层学习和深度学习

1. 浅层学习是机器学习的第一次浪潮

20 世纪 80 年代末期，用于人工神经网络的反向传播算法（也叫 Back Propagation 算法或者 BP 算法）的发明给机器学习带来了希望，掀起了基于统计模型的机器学习热潮。人们发现，利用 BP 算法可以让一个人工神经网络模型从大量训练样本中学习统计规律，从而对未知事件做出预测。这种基于统计的机器学习方法比起过去基于人工规则的系统，在很多方面显现出优越性。这个时候的人工神经网络虽然也被称作多层感知机（Multi-Layer Perceptron），但实际上是一种只含有一层隐层节点的浅层模型。

20 世纪 90 年代，各种各样的浅层机器学习模型相继被提出，如支持向量机（SVM）、Boosting、最大熵方法（如 LR）等。这些模型的结构基本上可以看作带有一层隐层节点（如 SVM、Boosting），或没有隐层节点（如 LR）。这些模型无论是在理论分析还是在实际应用中都获得了极大的成功。相比之下，由于理论分析的难度大，训练方法又需要很多经验和技巧，这个时期的浅层人工神经网络反而相对沉寂。

2. 深度学习是机器的第二次浪潮

2006 年，加拿大多伦多大学教授、机器学习领域的泰斗 Geoffrey Hinton 和他的学生 Ruslan Salakhutdinov 在《科学》杂志上发表了一篇文章，掀起了深度学习在学术界和工业界的浪潮。这篇文章有两个主要观点：（1）多隐层的人工神经网络具有优异的特征学习能力，学习得到的特征对数

据有更本质的刻画,从而有利于可视化或分类;(2)神经网络在训练上的难度可以通过“逐层初始化”来有效克服(在这篇文章中,逐层初始化是通过非监督式学习实现的)。

当前多数分类、回归等学习方法为浅层结构算法,其局限性在于,在有限样本和计算单元的情况下对复杂函数的表示能力有限,针对复杂分类问题其泛化能力受到一定制约。深度学习可以通过学习一种深层非线性网络结构,实现复杂函数逼近,表征输入数据分布式表示,并展现了强大的从少数样本集中学习数据集本质特征的能力,如图 8.16 所示。

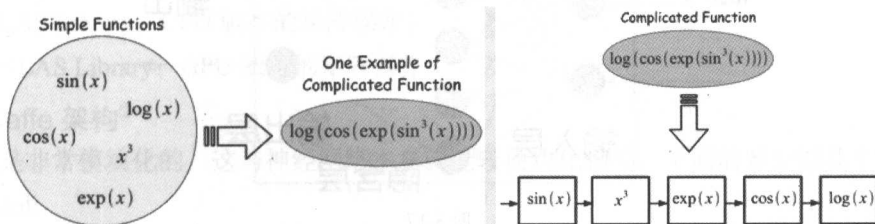


图 8.16

深度学习的实质是通过构建具有很多隐层的机器学习模型和海量的训练数据来学习更有用的特征,从而提升分类或预测的准确性。因此,“深度模型”是手段,“特征学习”是目的。区别于传统的浅层学习,深度学习的不同之处在于:(1)强调了模型结构的深度,通常有 5 层、6 层,甚至十几层的隐层节点;(2)明确突出了特征学习的重要性,也就是说,通过逐层特征变换,将样本在原空间的特征表示变换到一个新特征空间,从而使得分类或预测更加容易。与人工规则构造特征的方法相比,利用大数据来学习特征,更能够刻画出数据丰富的内在信息。

8.6.8 深度学习与神经网络

深度学习是机器学习研究中一个新的领域,其目的在于建立、模拟人脑进行分析学习的神经网络,它模仿人脑的机制来解释数据,如图像、声音和文本等。深度学习是非监督式学习的一种。

深度学习的概念来源于人工神经网络的研究。含多个隐层的多层感知器就是一种深度学习结构。深度学习通过组合低层特征形成更加抽象的高层特征来表示属性类别或特征,以发现数据的分布式特征表示。

深度学习本身是机器学习的一个分支,可以简单地理解为神经网络的发展。

二者的相同之处在于,深度学习采用了与神经网络相似的分层结构,系统由输入层、隐层(多层)、输出层组成,只有相邻的层节点之间有连接,同一层及跨层节点之间无连接,每一层都可以看作一个 Logistic Regression 模型,如图 8.17 所示。这种分层结构比较接近人脑的结构。

为了克服神经网络训练中的问题,深度学习采用了与神经网络不同的训练机制。在传统的神经网络中,采用迭代的算法来训练整个网络,随机设定初值,计算当前网络的输出,然后根据当前输出和 Label 之间的差去改变前面各层的参数,直到收敛(整体是一个梯度下降法)。而深度学习整体上是一个 layer-wise(逐层)的训练机制。这样做的原因是,如果采用 Back Propagation 的机制,那

么对于一个 Deep Network (7 层以上) 来说, 残差传播到最前面的层已经变得太小, 就会出现所谓的 Gradient Diffusion (梯度扩散)。

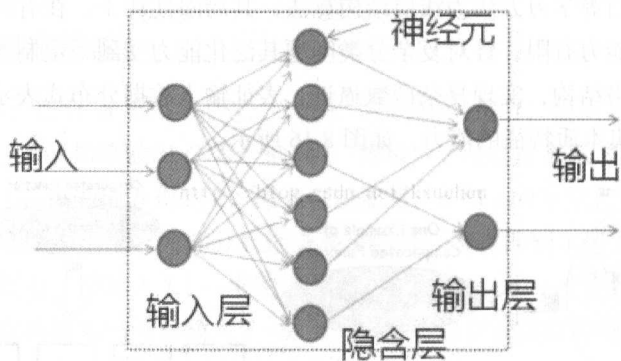


图 8.17

8.6.9 深度学习的训练过程

深度学习的训练过程具体如下:

(1) 自下而上的非监督式学习 (从底层开始, 一层一层地往顶层训练)。

采用无标定数据 (或有标定数据) 分层训练各层参数。这一步可以看作一个无监督训练过程, 是和传统神经网络区别最大的部分 (这个过程可以看作 Feature Learning 过程)。

具体而言, 先用无标定数据训练第一层, 训练时先学习第一层的参数 (这一层可以看作得到一个使得输出和输入差别最小的三层神经网络的隐层), 由于模型容量限制及稀疏性约束, 使得得到的模型能够学习到数据本身的结构, 从而得到比输入更具有表示能力的特征; 在学习得到第 $n-1$ 层后, 将第 $n-1$ 层的输出作为第 n 层的输入, 训练第 n 层, 由此分别得到各层的参数。

(2) 自顶向下的监督式学习 (通过带标签的数据去训练, 误差自顶向下传输, 对网络进行微调)。

基于第一步得到的各层参数进一步微调整个多层模型的参数, 这一步是一个有监督训练过程。第一步类似神经网络的随机初始化初值过程, 由于深度学习的第一步不是随机初始化, 而是通过学习输入数据的结构得到的, 因而这个初值更接近全局最优, 从而能够取得更好的效果。所以深度学习效果好很大程度上归功于第一步的 Feature Learning 过程。

8.6.10 深度学习的框架

深度学习还处于一个蓬勃发展的阶段, 各种深度学习框架层出不穷。常见的深度学习框架有 Tensorflow、Caffe、Theano、Torch、Deeplearning4j、Marvin、ConvNetJS 和 MXNet 等。

下面来看一下 Caffe 架构, 以便对深度学习框架有一个基本的认识。

1. Caffe^①背景

Caffe 是一个清晰而高效的深度学习框架，它是纯粹的 C++/CUDA 架构，支持命令行、Python 和 MATLAB 接口，可以在 CPU 和 GPU 之间无缝切换。

Caffe 中用到的主要的库有如下几个。

- Google Logging Library (Glog): 一个 C++ 语言的应用级日志记录框架。
- LebelDB (数据存储): 是一个 Google 实现的非常高效的 KV 数据库，单进程操作。
- CBLAS Library: CPU 版本的矩阵操作。
- CUBLAS Library: GPU 版本的矩阵操作。

2. Caffe 架构^②

Caffe 是非常模块化的，这与神经网络本身就比较模块化相关。下面简要介绍几个常用的模块。

1) Blob

Blob 是一个四维连续数组，如果使用 (n, c, h, w) 表示的话，那么每一维的意思分别如下。

- n (number): 输入数据量。
- c (channel): 如果是图像数据，则可以认为是通道数量。
- h, w (height, width): 如果是图像数据，则可以认为是图片的高度和宽度。

当然，Blob 不一定就是用来表示图像输入数据的。

Blob 内部有两个字段：data 和 diff。data 表示流动数据（输出数据），而 diff 则存储 BP 的梯度。data/diff 可以存储于 CPU，也可以存储于 GPU。如果某个 Layer 不支持 GPU，那么就需要将 GPU 数据复制到 CPU 上，从而造成性能开销。对于 Python/NumPy 用户来说，可以用 reshape() 函数来转换为 Blob: `data = data.reshape((-1, c, h, w))`。

2) Layer

Caffe 提供了许多内置 Layer，每个 Layer 的输入称为 bottom blob，输出称为 top blob，如图 8.18 所示。

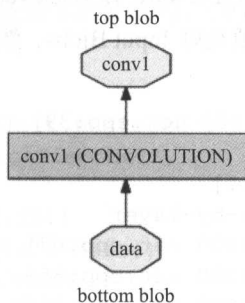


图 8.18

① Caffe 框架，<http://caffe.berkeleyvision.org/>。

② 参考 <http://dirlt.com/caffe.html>。

每层定义三个主要的计算过程：初始化、前向传播、反向传播/计算梯度。

3) Net

Net 是由 Layers 组成的 DAG, 并且可以使用文本格式来描述。下面文本生成的是 Logistic Regression。

```
name: "LogReg"
layers {
  name: "mnist"
  type: DATA
  top: "data"
  top: "label"
  data_param {
    source: "input_leveldb"
    batch_size: 64
  }
}
layers {
  name: "ip"
  type: INNER_PRODUCT
  bottom: "data"
  top: "ip"
  inner_product_param {
    num_output: 2
  }
}
layers {
  name: "loss"
  type: SOFTMAX_LOSS
  bottom: "ip"
  bottom: "label"
  top: "loss"
}
```

Net 有一个初始化函数 Init(), 它有两个作用: (1) 创建 Blobs 和 Layers; (2) 调用 Layers 的 SetUp 函数来初始化 Layers。Net 还有两个函数 Forward 和 Backward, 分别调用各个 Layers 的 Forward 和 Backward。如果需要进行预测, 则需先填充好 Input Blobs, 然后调用 Forward 函数, 最后获取 Output Blobs 作为预测结果。代码如下:

```
I0902 22:52:17.931977 2079114000 net.cpp:39] Initializing net from parameters:
name: "LogReg"
[...model prototxt printout...]
# construct the network layer-by-layer
I0902 22:52:17.932152 2079114000 net.cpp:67] Creating Layer mnist
I0902 22:52:17.932165 2079114000 net.cpp:356] mnist -> data
I0902 22:52:17.932188 2079114000 net.cpp:356] mnist -> label
I0902 22:52:17.932200 2079114000 net.cpp:96] Setting up mnist
I0902 22:52:17.935807 2079114000 data_layer.cpp:135] Opening leveldb input_leveldb
I0902 22:52:17.937155 2079114000 data_layer.cpp:195] output data size: 64,1,28,28
I0902 22:52:17.938570 2079114000 net.cpp:103] Top shape: 64 1 28 28 (50176)
```



```

I0902 22:52:17.938593 2079114000 net.cpp:103] Top shape: 64 1 1 1 (64)
I0902 22:52:17.938611 2079114000 net.cpp:67] Creating Layer ip
I0902 22:52:17.938617 2079114000 net.cpp:394] ip <- data
I0902 22:52:17.939177 2079114000 net.cpp:356] ip -> ip
I0902 22:52:17.939196 2079114000 net.cpp:96] Setting up ip
I0902 22:52:17.940289 2079114000 net.cpp:103] Top shape: 64 2 1 1 (128)
I0902 22:52:17.941270 2079114000 net.cpp:67] Creating Layer loss
I0902 22:52:17.941305 2079114000 net.cpp:394] loss <- ip
I0902 22:52:17.941314 2079114000 net.cpp:394] loss <- label
I0902 22:52:17.941323 2079114000 net.cpp:356] loss -> loss
# set up the loss and configure the backward pass
I0902 22:52:17.941328 2079114000 net.cpp:96] Setting up loss
I0902 22:52:17.941328 2079114000 net.cpp:103] Top shape: 1 1 1 1 (1)
I0902 22:52:17.941329 2079114000 net.cpp:109] with loss weight 1
I0902 22:52:17.941779 2079114000 net.cpp:170] loss needs backward computation.
I0902 22:52:17.941787 2079114000 net.cpp:170] ip needs backward computation.
I0902 22:52:17.941794 2079114000 net.cpp:172] mnist does not need backward
computation.
# determine outputs
I0902 22:52:17.941800 2079114000 net.cpp:208] This network produces output loss
# finish initialization and report memory usage
I0902 22:52:17.941810 2079114000 net.cpp:467] Collecting Learning Rate and Weight
Decay.
I0902 22:52:17.941818 2079114000 net.cpp:219] Network initialization done.
I0902 22:52:17.941824 2079114000 net.cpp:220] Memory required for data: 201476

```

如果阅读 Caffe/Models 就会发现, 这些例子下面有 train.prototxt 和 deploy.prototxt, 差别仅仅在于 deploy.txt 没有 data-layer^①, 而是指定输入的 shape^②。

```

input: "data"
input_dim: 10
input_dim: 1
input_dim: 28
input_dim: 28

```

从字面上来看, train.prototxt 是用来训练出模型的, 而 deploy.prototxt 则是用来进行预测的。如下是使用 Python 进行预测的代码:

```

caffe.set_mode_cpu()
net = caffe.Net('caffe-conf/test.prototxt',
                'uv_iter_10000.caffemodel',
                caffe.TEST)
data = data.reshape((-1, 1, 28, 28))
out = net.forward_all(**{'data': data})
rs = out['prob'] # 得到的是 softmax
print_timer("predict")

```

① data_layer: 网络的底层, 主要用于将数据经 Blob 传入网络中。

② Shape: 存放形状的数据结构。

4) Solver

下面是 solver.prototxt 的一个示例（从 examples/mnist/修改而来）：

```
# The train/test net protocol buffer definition
net: "caffe-conf/train.prototxt"
# 如果 test 数据量是 10000, 而 batch_size = 100, 那么 test_iter 就应该设置为 100
# 这样每次进行 test 就可以使用所有的 cases
test_iter: 90
# Carry out testing every 500 training iterations.
# 每进行 500 轮迭代进行一次测试
test_interval: 500
# 下面这些是训练所用的参数
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
# The learning rate policy
lr_policy: "inv"
gamma: 0.0001
power: 0.75
# Display every 100 iterations
display: 500
# The maximum number of iterations
max_iter: 10000
# snapshot intermediate results
# 每进行 500 轮迭代进行一次 snapshot
# 每一轮使用的数据量大小为 batch_size
snapshot: 500
snapshot_prefix: "uv"
snapshot_after_train: true
# solver mode: CPU or GPU
# 使用 CPU 训练
solver_mode: CPU
```

“net”表示 train 和 test 使用同一个 Net。在 net.prototxt 中可以使用 include 语法来声明某个 Layer 是否需要包含在 train/test 阶段。

如果在训练时不想进行 test, 那么可以指定上面的“net”为“train_net”。当然也可以使用“test_nets”来指定多个 test_net。

5) Python 接口^①

Caffe 支持三种接口：命令行、Python 和 MATLAB。这里只介绍 Python 接口。

^① <http://caffe.berkeleyvision.org/tutorial/interfaces.html>

- caffe.Net: 加载、配置和执行模型的主要接口。
- caffe.Classifier 和 caffe.Detector: 给普通任务提供方便的接口。
- caffe.io: 负责预处理输入、输出和通信缓存。
- caffe.draw: 提供网络架构的可视化。
- Caffe blobs: 提供多维矩阵易用接口。

3. Caffe 使用案例: 使用数据库 CIFAR-10^①

60 000 张 32 像素×32 像素的彩色图像 10 类, 50 000 张用于训练, 10 000 张用于测试, 如图 8.19 所示。



图 8.19

1) 准备

在终端运行以下指令:

```
cd $CAFFE_ROOT/data/cifar10
./get_cifar10.sh
cd $CAFFE_ROOT/examples/cifar10
./create_cifar10.sh
```

其中, CAFFE_ROOT 是 Caffe-Master 在主机上的地址。

运行之后, 将会在 examples 目录中出现数据库文件 ./cifar10-leveldb 和数据库图像均值二进制文件 ./mean.binaryproto, 如图 8.20 所示。

^① 案例 <http://caffe.berkeleyvision.org/gathered/examples/cifar10.html>。

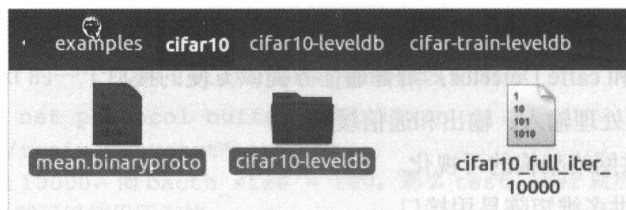


图 8.20

2) 模型

该 CNN 由卷积层、POOLing 层、非线性变换层、在顶端的局部对比归一化线性分类器组成。该模型的定义在 `CAFFE_ROOT/examples/cifar10_quick_train.prototxt` 文件中，可以进行修改。

3) 训练和测试

训练这个模型非常简单，在写好参数设置的文件 `cifar10_quick_solver.prototxt` 及定义的文件 `cifar10_quick_train.prototxt` 和 `cifar10_quick_test.prototxt` 以后，运行 `train_quick.sh`，或者在终端输入如下命令即可：

```
cd $CAFFE_ROOT/examples/cifar10
./train_quick.sh
```

`train_quick.sh` 是一个简单的脚本，它会把执行的信息显示出来。训练的工具是 `train_net.bin`，以 `cifar10_quick_solver.prototxt` 作为参数。然后出现类似如下信息：

```
I0317 21:52:48.945710 2008298256 net.cpp:74] Creating Layer conv1
I0317 21:52:48.945716 2008298256 net.cpp:84] conv1 <- data
I0317 21:52:48.945725 2008298256 net.cpp:110] conv1 -> conv1
I0317 21:52:49.298691 2008298256 net.cpp:125] Top shape: 100 32 32 32 (3276800)
I0317 21:52:49.298719 2008298256 net.cpp:151] conv1 needs backward computation.
```

接着：

```
O317 21:52:49.309370 2008298256 net.cpp:166] Network initialization done.
I0317 21:52:49.309376 2008298256 net.cpp:167] Memory required for Data 23790808
I0317 21:52:49.309422 2008298256 solver.cpp:36] Solver scaffolding done.
I0317 21:52:49.309447 2008298256 solver.cpp:47] Solving CIFAR10_quick_train
```

之后，训练开始：

```
I0317 21:53:12.179772 2008298256 solver.cpp:208] Iteration 100,lr = 0.001
I0317 21:53:12.185698 2008298256 solver.cpp:65] Iteration 100,loss = 1.73643
...
I0317 21:54:41.150030 2008298256 solver.cpp:87] Iteration 500,Testing net
I0317 21:54:47.129461 2008298256 solver.cpp:114] Test score #0: 0.5504
I0317 21:54:47.129500 2008298256 solver.cpp:114] Test score #1: 1.27805
```

其中，每 100 次迭代显示一次训练时 `lr` (learningrate) 和 `loss` (训练损失函数)，每 500 次测试一次，输出 `score 0` (准确率) 和 `score 1` (测试损失函数)。当 5000 次迭代之后，准确率约为 75%，模型的参数以二进制 Protobuf 格式存储在 `cifar10_quick_iter_5000` 中，然后这个模型就可以用来运行在新数据上了。

4) 对比 CPU/GPU

另外, 更改 `cifar*solver.prototxt` 文件可以使用 CPU 训练, 可以对比 CPU 训练和 GPU 训练的差别。

```
# solver mode: CPU or GPU
solver_mode: CPU
```

8.6.11 深度学习与 GPU

深度学习通过构建深层神经网络来模拟人脑的工作原理。如图 8.21 所示, 深层神经网络由一个输入层、数个隐层及一个输出层构成。每层有若干个神经元, 神经元之间有连接权重。每个神经元模拟人类的神经细胞, 而节点之间的连接模拟神经细胞之间的连接。

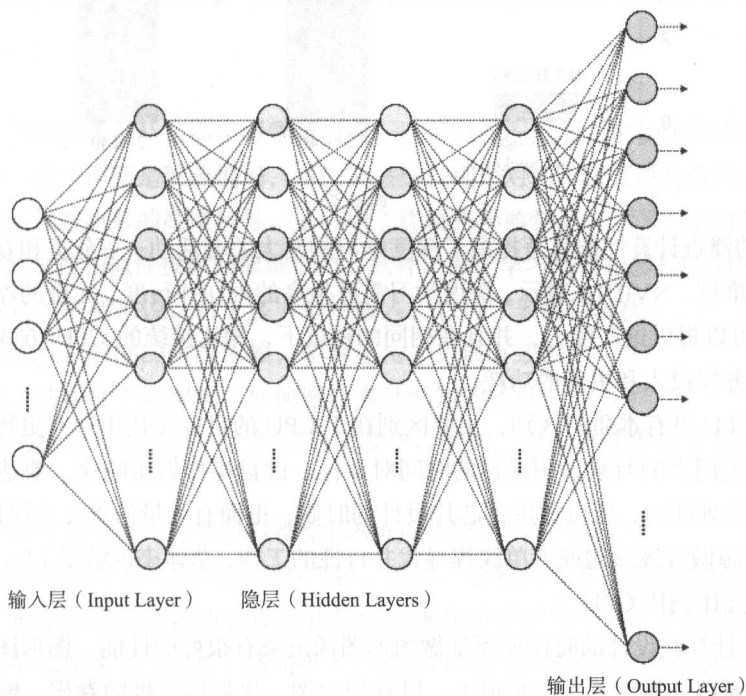


图 8.21

深层神经网络模型复杂, 训练数据多, 计算量大。一方面, DNN 需要模拟人脑的计算能力, 而人脑包含 100 多亿个神经细胞, 这要求 DNN 中的神经元非常多, 神经元间的连接数量也相当惊人。从数学的角度看, DNN 中的每个神经元都包含数学计算 (如 Sigmoid、ReLU 或者 Softmax 函数), 需要估计的参数数量也极大。在语音识别和图像识别应用中, 神经元达数十万个, 参数达数千万个, 模型复杂导致计算量大。另一方面, DNN 需要大量数据才能训练出高准确率模型。DNN 参数量大、模型复杂, 为了避免过拟合, 需要海量训练数据。两方面因素叠加, 导致训练一个模型耗时惊人。以语音识别为例, 目前业界通常使用的样本量达数十亿, 用 CPU 单机需要数年才能完成一次训练, 用流行的 GPU 卡也需要数周才能完成一次训练。

图 8.22 是 CPU 和 GPU 对比基准测试，相比 CPU，借助 GPU 执行任务的速度可以提升几十倍。

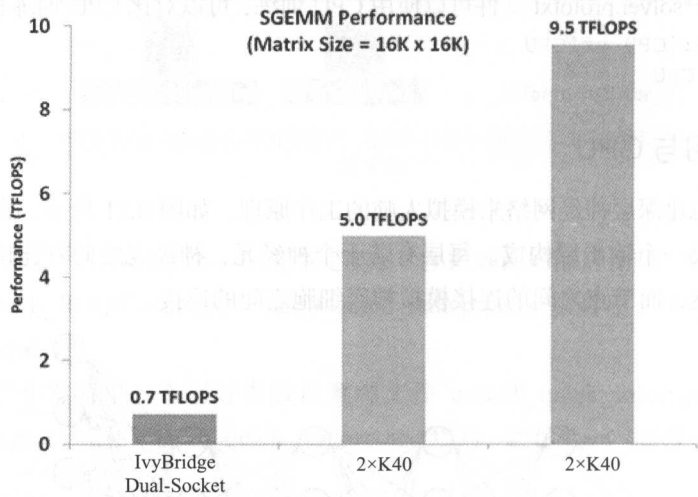


图 8.22

GPU 出色的浮点计算性能显著提高了深度学习的两大关键活动——分类和卷积的性能，同时又达到了所需的精准度。NVIDIA 表示，深度学习需要很高的内在并行度、大量的浮点计算能力及矩阵预算，而 GPU 可以提供这些能力，并且在相同的精度下，相比传统的 CPU 方式，拥有更快的处理速度、更少的服务器投入和更低的功耗。

而 GPU 和 CPU 没有本质的区别，主要区别在于 CPU 的目标是让用户有更短的响应时间，即在编辑文档或者浏览网页的时候，用最短的时间对鼠标、键盘操作做出响应。要达到这个目的，最重要的是单线程的处理能力，所以在进行芯片设计的时候，里面有大量的单元来保证单线程处理性能。CPU 有大量的资源做分级预测或者单线程寻找并行性的工作，总体来说就是 LU，即真正实现浮点整行运算的公共单元比例比 GPU 少。

另外，GPU 计算在设计的时候要保证做图形图像渲染有很好的性能。图形图像渲染任务与其他领域的计算或者通用计算的差别不是很大。以卷积为例，比如做二维的卷积，输入的是图像，竖直的角度是一个矩阵，卷积核每个元素和输入图像进行相应的乘加，放在输出的二维矩阵里。

除了 GPU 外，为了解决深度学习计算效率的问题，业界还有一种思路，即打造深度学习专用芯片。最典型的是 IBM 研发出一种新型芯片 SyNAPSE，它虽然拥有 100 万个神经元芯片、2.56 亿个突触、4096 个核心及 54 亿个晶体管，但它的功率仅有 63mW。这种新芯片可以伸缩扩展，数个芯片连接在一起就可以组成强大的神经网络。

8.6.12 深度学习小结与展望

1. 小结

深度学习算法自动提取分类所需的低层次或者高层次特征。高层次特征是指该特征可以分级（层

次)地依赖其他特征。例如,对于机器视觉,深度学习算法从原始图像去学习得到它的一个低层次表达,如边缘检测器、小波滤波器等,然后在这些低层次表达的基础上再建立表达,如这些低层次表达的线性或者非线性组合,然后重复这个过程,最后得到一个高层次的表达。

深度学习能够更好地表示数据的特征,同时由于模型的层次、参数很多,容量足够,因此,模型有能力表示大规模数据。所以对于图像、语音这种特征不明显(需要手工设计且很多没有直观的物理含义)的问题,能够在大规模训练数据上取得更好的效果。此外,从模式识别特征和分类器的角度来看,深度学习框架将特征和分类器结合到一个框架中,用数据去学习特征,在使用中减少了手工设计特征的巨大工作量,因此,不仅效果更好,而且使用起来也有很多方便之处。

当然,深度学习本身并不是完美的,也不是解决任何机器学习问题的利器,不应该被放大到一个无所不能的程度。

2. 未来

深度学习目前的关注点还是从机器学习领域借鉴一些可以在深度学习中使用的方法,特别是降维领域。例如,目前的一项工作就是稀疏编码,通过压缩感知理论对高维数据进行降维,使用元素非常少的向量就可以精确地代表原来的高维信号。另一项工作就是半监督流行学习,通过测量训练样本的相似性,将高维数据的这种相似性投影到低维空间。此外,深度学习还有很多核心问题需要解决:

- (1) 对于一个特定的框架来说,对于多少维的输入,它可以表现得较优?
- (2) 对捕捉短时或者长时间的时间依赖,哪种架构才是最有效的?
- (3) 对于一个给定的深度学习架构,如何融合多种感知的信息?
- (4) 有什么正确的机理可以增强一个给定的深度学习架构,以改进其鲁棒性及对扭曲和数据丢失的不变性?
- (5) 模型方面是否有其他更为有效且有理论依据的深度模型学习算法?

探索新的特征提取模型是值得深入研究的内容。此外,有效的可并行训练算法也是值得研究的一个方向。当前基于最小批处理的随机梯度优化算法很难在多计算机中进行并行训练,通常的解决办法是利用图形处理单元加速学习过程。然而单个机器 GPU 对大规模数据识别或相似任务数据集并不适用。在深度学习的应用拓展方面,如何合理且充分地利用深度学习增强传统学习算法的性能仍是目前各领域的研究重点。

8.7 小结

本章主要介绍了机器学习,尤其是当前最热门的深度学习。深度学习可以说掀起了人工智能的又一次热潮,但是大家要清楚地认识到,这离真正的 AI(人工智能)还差得很远。但总的来说,我们离电影中描述的未来世界更近了一步,不是吗?

第 9 章

资源管理

资源管理的本质是集群、数据中心级别资源的统一管理和分配，以提高效率。其中，多租户、弹性伸缩、动态分配是资源管理系统要解决的核心问题。

本章首先介绍资源管理的基本概念，以及 Hadoop 领域常见的一些资源调度框架；接着介绍大数据时代面临的多租户问题、数据中心的统一资源调度、资源调度和分配算法，以及基于应用描述的智能调度；最后介绍一个 Mesos 代码分析实战。

9.1 资源管理的基本概念

9.1.1 资源调度的目标 and 价值

1. 子系统高效调度

任务之间资源隔离，减少争抢。任务分配调度时结合资源分配，各个任务分配合理的资源，充分利用系统资源，减少资源利用不充分的问题。资源调度结合优先级，优先级高的可以分配更多的资源。

2. 提高全系统的资源利用率

各个子系统存在不同时期对资源需求不一样的情况，平滑系统资源的利用。

3. 支持动态调整切分资源，增强系统扩展性

子系统对资源的规划很难一次性确定，通过资源调度管理系统支持动态扩展，增强系统扩展性。

9.1.2 资源调度的使用限制及难点

1. 实时业务受限

资源调度是为了提高资源利用率，分配本身是存在一定的开销的，对实时性要求非常高的应用不适合（毫秒、秒级别的应用）。

2. 应用（框架）资源规划难

资源框架通过算法分配资源，但是每个细粒度的具体的任务对资源的需求很难预估。规划如果偏差较大，反而会降低系统本身的性能。

3. 内存使用分配难

JVM 虚拟机存在内存回收的问题，上层应用程序无法控制。内存很难分配准确，如果内存分配过少，则会导致任务失败；分配过多，则会造成资源浪费。

9.2 Hadoop 领域的资源调度框架

Hadoop 开源领域比较有名的资源调度框架有 YARN、Mesos、Borg，以及研究项目 Omega 等，本节介绍各种框架的架构和优缺点。

9.2.1 YARN

1. 背景

YARN 从 Hadoop 1.0 发展而来，解决了 Hadoop 1.0 的两个主要问题：

(1) 单管理节点性能瓶颈。一个管理节点能管理的服务器不能无上限。

(2) Hadoop 1.0 按照 slot 来划分资源，mapslot 的资源不能共享给 reduceslot，造成资源浪费。很多公司目前切换到 Hadoop 2.0，如淘宝天梯已经淘汰 1.0，上线 2.0。

2. 架构

YARN 的架构如图 9.1 所示。

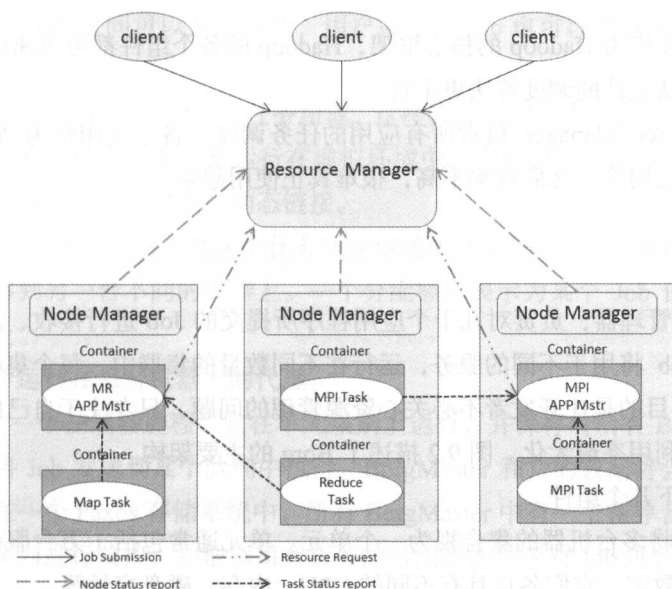


图 9.1

MapReduce v2 最基本的设计思想是将 JobTracker 的两个主要功能，即资源管理和作业调度/监控分成两个独立的进程。在该解决方案中包含两个组件：全局的 Resource Manager (RM) 和与每个应

用相关的 Application Master (AM)。这里的“应用”是指一个单独的 MapReduce 作业或者 DAG 作业。RM 和 Node Manager (NM, 每个节点一个) 共同组成整个数据计算框架。RM 是系统中将资源分配给各个应用的最终决策者。AM 实际上是一个具体的框架库, 它的任务是与 RM 协商获取应用所需的资源, 以及与 NM 合作, 完成执行和监控 Task 的任务。

调度器根据容量、队列等限制条件(如每个队列分配一定的资源、最多执行一定数量的作业等), 将系统中的资源分配给各个正在运行的应用。这里的调度器是一个“纯调度器”, 因为它不再负责监控或者跟踪应用的执行状态等, 也不负责重新启动因应用执行失败或者硬件故障而产生的失败任务。调度器仅根据各个应用的资源需求进行调度, 这是通过抽象概念“资源容器”完成的。资源容器(Resource Container)将内存、CPU、磁盘、网络等资源封装在一起, 从而限定每个任务使用的资源量。

调度器是可插拔的组件, 主要负责将集群中的资源分配给多个队列和应用。YARN 自带多个资源调度器, 如 Capacity Scheduler 和 Fair Scheduler 等。

ASM 主要负责接收作业, 协商获取第一个容器用于执行 AM 和提供重启失败 Application Container 的服务。

NM 是每个节点上的框架代理, 主要负责启动应用所需的容器, 监控资源(内存、CPU、磁盘、网络等)的使用情况并将其汇报给调度器。

AM 主要负责同调度器协商以获取合适的容器, 并跟踪这些容器的状态和监控其进度。

3. 优缺点

(1) 优点: YARN 作为 Hadoop 的核心框架, Hadoop 的各个组件都可快速地接入 YARN 框架, 未来发展会很快, 默认支持的调度算法更丰富。

(2) 缺点: Resource Manager 负责所有应用的任务调度, 各个应用作为 YARN 的一个 Client Library。传统数据库应用接入之后效率不高, 很难真正使用起来。

9.2.2 Borg^①

Borg 是一个集群管理器, 负责对几千个应用程序所提交的 Job 进行接收、调试、启动、停止、重启和监控, 这些 Job 将用于不同的服务, 运行在不同数量的集群中, 每个集群各自可包含最多几万台服务器。Borg 的目的是让开发者不必关心资源管理的问题, 只专注于自己的工作, 并且做到跨多个数据中心的资源利用率最大化。图 9.2 描述了 Borg 的主要架构。

该架构中包含以下几个组件。

- 单元(Cell): 将多台机器的集合视为一个单元。单元通常包括 1 万台服务器, 如果有必要也可以增加这个数字, 它们各自具有不同的 CPU、内存、磁盘容量等。

^① 参见 <http://www.infoq.com/cn/news/2015/04/google-borg/>。

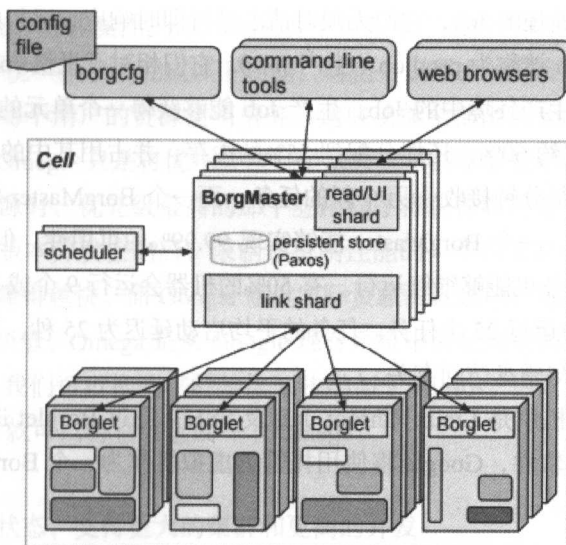


图 9.2

- 集群：一般来说包含一个大型单元，有时也会包含一些用于特定目的的小单元，其中有些单元可以用作测试。一个集群通常来说限制在一个数据中心大楼里，集群中的所有机器都是通过高性能的网络进行连接的。一个网站可以包含多个大楼和集群。
- Job：一种在单元的边界之内执行的活动。这些 Job 可以附加一些需求信息，如 CPU、OS、公开的 IP 等。Job 之间可以互相通信，用户或监控 Job 也可以通过 RPC 方式向某个 Job 发送命令。
- Task：一个 Job 可以由一个或多个任务组成，这些任务在同一个可执行进程中运行。这些任务通常直接运行在硬件上，而不是运行在虚拟环境中，以避免虚拟化的成本。任务的程序是静态链接的，以避免在运行时进行动态链接。
- 分配额 (Alloc)：专门为一个或多个任务所保留的机器资源集。分配额能够与运行于其上的任务一起被转移到另一台不同的机器上。一个分配额集表示为某个 Job 保留的资源，并且分布多台机器上。
- Borglet：一个运行在每台机器上的代理。
- BorgMaster：一个控制器进程，它在单元级别上运行，并保存着所有 Borglet 上的状态数据。BorgMaster 将 Job 发送到某个队列中执行。BorgMaster 和它的数据将会进行 5 次复制，数据将被持久化在一个 Paxos 存储系统中。所有 BorgMaster 中有一个领导者。
- 调度器：对队列进行监控，并根据每台机器的可用资源情况对 Job 进行调度。

Borg 系统的使用者将向系统提交包含一个或多个任务的 Job，这些任务将共享同样的二进制代码，并在一个单元中执行，每个 Borg 单元由多台机器组成。在这些单元中，Borg 会组合两种类型的活动：一种是如 Gmail、GDocs、BigTable 之类的长期运行服务，这些服务的响应延迟很短，最多为

几百毫秒；另一种是批量处理的 Job，它们无须对请求进行即时响应，运行的时间可能会很长，甚至是几天。第一种类型的 Job 被称为 prodjob（生产 Job），它们相对于批量 Job 来说优先级更高；第二种类型的 Job 被认为是非生产环境中的 Job。生产 Job 能够获得一个单元的 CPU 资源中的 70%，并且占用所有 CPU 数量的大约 60%，还能分配到 55% 的内存，并占用其中的大约 85%。

某些单元的任务量是每分钟接收 1 万个新的任务，而一个 BorgMaster 能够使用 10~14 个 CPU 内核，以及 50GB 的内存。一个 BorgMaster 能够实现 99.99% 的可用性，但即使某个 BorgMaster 或 Borglet 出现停机状况，任务也能够继续运行。有 50% 的机器会运行 9 个或 9 个以上的任务，而某些机器能够在 4500 个线程中运行 25 个任务。任务的平均启动延迟为 25 秒，其中的 20 秒用于安装包。这些等待时间中的大部分与磁盘访问有关。

这套系统的主要安全机制是 Linux Chroot Jail 及 SSH，通过 Borglet 进行任务调试。对于运行在 GAE 或 GCE 中的外部软件，Google 将使用托管的虚拟机作为一个 Borg 任务在某个 KVM 进程中运行。

9.2.3 Omega

1. 背景

Google 的论文 *Omega: flexible, scalable schedulers for large compute clusters* 中把调度分为 3 代：第一代是独立的集群；第二代是两层调度（Mesos，YARN）；第三代是共享状态调度，如图 9.3 所示。

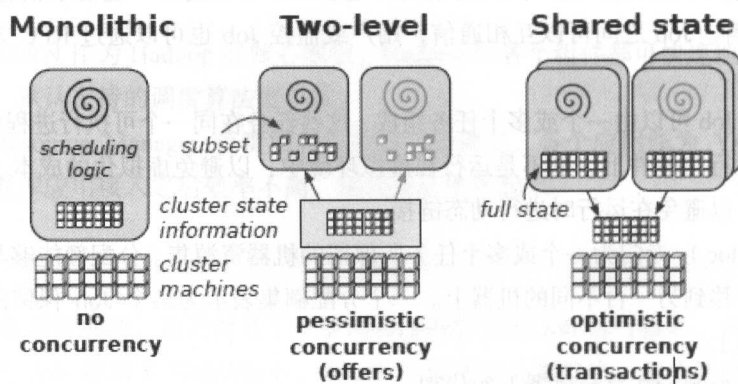


图 9.3

2. 架构

为了克服双层调度器的局限性，Google 开发了下一代资源管理系统 Omega。Omega 是一种基于共享状态的调度器，该调度器将双层调度器中的集中式资源调度模块简化为一些持久化的共享数据（状态）和针对这些数据的验证代码，而这里的“共享数据”实际上就是整个集群的实时资源使用信息。一旦引入共享数据，共享数据的并发访问方式就成为该系统设计的核心。而 Omega 则采用传统数据库中基于多版本的并发访问控制方式（也称为“乐观锁”，Multi-Version Concurrency Control，

MVCC), 大大提升了 Omega 的并发性。由于 Omega 不再有集中式的调度模块, 因此, 不能像 Mesos 或者 YARN 那样, 在一个统一模块中完成以下功能: 对整个集群中的所有资源分组; 限制每类应用程序的资源使用量; 限制每个用户的资源使用量等。这些功能全部由各个应用程序调度器自我管理和控制。根据论文所述, Omega 只是将优先级这一限制放到了共享数据的验证代码中, 即当多个应用程序同时申请同一份资源时, 优先级最高的那个应用程序将获得该资源, 其他资源限制全部下放到各个子调度器。引入多版本并发控制后, 限制该机制性能的一个因素是资源访问冲突的次数, 冲突次数越多, 系统性能下降得越快。而 Google 通过实际负载测试证明, 这种方式的冲突次数是完全可以接受的。该论文中还谈道, Omega 是从 Google 现有系统中演化而来的。虽然这篇论文只介绍了 Omega 的调度器架构, 但我们可以推测它的整体架构类似于 Mesos。因此, 如果你了解 Mesos, 则通过修改 Mesos 的 Master 就可以将它改造成一个 Omega。

3. 优缺点

(1) 优点: 共享资源状态, 支持更大的集群和更高的并发。

(2) 缺点: 只有论文, 无具体实现, 在小集群下没有优势。

9.2.4 本节小结

除了 Mesos、YARN、Borg、Omega 外, 业界常见的资源调度框架还有 Hadoop Corona, 它是 Facebook 开源的下一代 MapReduce 框架, 其基本设计动机和 Apache 的 YARN 一致; 腾讯的 Torca 参考了 YARN 的很多实现, 在资源分配和利用上有一些独特的地方; Borg 中的很多思路被应用到 Google 新开源的 Kubernetes 软件中去。

9.3 资源分配算法

9.3.1 算法的作用

资源管理和分配的核心是资源的分配, 这就涉及公平、效率、优先级等问题。资源管理算法就是通过算法来均衡几者的关系, 算法的效率、能力至关重要, 决定了资源调度系统的能力。

9.3.2 几种调度算法分析

1. Mesos 调度算法

Mesos 采用 Dominant Resource Fair (DRF) 算法进行资源分配时分为两层, 第一层由 Master 负责, 将资源分配给 Framework。DRF 算法是多资源环境下的最大最小公平 (Max-Min Fair) 算法, 该算法通过为每个用户定义一个 Dominant Share, 并根据 Dominant Share 进行公平分配以实现将多资源分配转化为单资源分配问题。其中, Dominant Share 为用户对资源的占用率最大的值, 其定义为

$s_i = \max\{u_{ij}/r_j\}$ (u_{ij} 表示用户 i 对资源 j 的占用量, r_j 表示资源 j 的总量)。Dominant Share 对应的资源为 Dominant Source (占用总资源百分比最大的那个资源)。

DRF 算法的思想是: 首先, 计算每个用户的每种资源的占用率 (Share), 选择占用率中的最大值为用户的 Dominant Share s_i 。其次, 每次从拥有最低的 Dominant Share 的用户中选择一个任务准备运行, 若系统中有足够的可用资源执行该任务, 则启动该任务。重复上述过程, 直到不存在可用资源或不存在需要执行的任务。

DRF 算法伪代码如下:

Algorithm 1 DRF pseudo-code

$R = \langle r_1, \dots, r_m \rangle$ \triangleright total resource capacities
 $C = \langle c_1, \dots, c_m \rangle$ \triangleright consumed resources, initially 0
 s_i ($i = 1..n$) \triangleright user i 's dominant shares, initially 0
 $U_i = \langle u_{i,1}, \dots, u_{i,m} \rangle$ ($i = 1..n$) \triangleright resources given to user i , initially 0

pick user i with lowest dominant share s_i

$D_i \leftarrow$ demand of user i 's next task

if $C + D_i \leq R$ **then**

$C = C + D_i$ \triangleright update consumed vector

$U_i = U_i + D_i$ \triangleright update i 's allocation vector

$s_i = \max_{j=1}^m \{u_{i,j}/r_j\}$

else

return \triangleright the cluster is full

end if

实例 1: 考虑一个 9CPU、18GB RAM 的系统, 拥有两个用户, 其中用户 A 运行的任务的需求向量为 {1CPU, 4GB}, 用户 B 运行的任务的需求向量为 {3CPU, 1GB}。

采用 DRF 算法进行资源分配的过程如下: DRF 首先选择用户 B 来运行一个任务, 结果用户 B 的资源占用率变为 {3/9, 1/18}, 用户 B 的 Dominant Share 变为 $\max\{3/9, 1/18\} = 1/3$ 。接下来 DRF 选择用户 A, 因为此时用户 A 的资源占用率为 0。这个过程持续进行, 直到不再可能运行新任务。在这种情况下, 一般会出现 CPU 饱和。在以上分配过程的最后, 用户 A 会得到 {3CPU, 12GB}, 用户 B 会得到 {6CPU, 2GB}, 每个用户都会获得 2/3 的 Dominant Resource。

DRF 算法主要分为不带权 DRF 和带权 DRF, 上面描述的是不带权 DRF。DRF 的主要不足为:

(1) DRF 算法过度追求公平, 未考虑实际应用问题。

(2) 资源碎片问题。Mesos 采用了 Resource Offer 机制, 这种调度机制面临着资源碎片问题, 即每个节点上的资源不可能全部被分配完, 剩下的资源可能不足以让任何任务运行, 这样便产生了类似操作系统中的内存碎片问题。

(3) 分配策略限定在 DRF 算法内, 不够丰富。

(4) 不支持框架对资源进行抢占, 框架无法获取各个集群的状态。

1) Mesos DRF 介绍

Mesos 调度器是根据 DRF (Dominant Resource Fairness) 算法实现的。DRF 算法背后的直观想法

是：在多资源类型的环境下，一个用户的资源分配应该由用户的 Dominant Share（主导份额的资源）决定，Dominant Share 是在所有已经分配给用户的多种资源中占据最大份额的一种资源。

Mesos 使用 Resource Offer 的方式实现各个框架之间的资源分配。Resource Offer 是多个 Slave 节点上的一组空闲资源。Master 根据调度策略来决定提供多少资源给每个 Framework，通过 Resource Offer 的形式发送给框架，然后框架响应 Resource Offer，确认 Resource Offer 中已使用的资源，并返回剩余的空闲资源。

Mesos 的资源分配器是一个分层的加权最大最小公平算法的实现，通过抽象一个 Role 概念，将 Framework 按照 Role 进行分组。这样一来，资源分配就可以分为两层：首先是在各个框架群组间通过加权的 DRF 算法进行排序；其次是在框架群组内部对各个 Framework 使用加权 DRF 算法进行排序。然后按照最终的排序结果，从小到大对各个框架进行 Resource Offer。

2) Mesos 资源分配器的实现

Mesos 资源分配器的实现分为两部分：Allocator 和 Sorter。Allocator 定义和实现了资源分配器的接口和逻辑；Sorter 使用具体的资源分配算法来对资源使用者进行排序。

Sorter 的实现是 DRFSorter，DRFSorter 使用 DRF 算法来对资源使用者进行排序。Allocator 的实现是 HierarchicalAllocatorProcess，它实现了一个分层的分配器，对 Framework 进行分组，分别在各组之间和组内部使用 DRFSorter 进行排序。

3) DRFSorter

DRFSorter 通过每个用户的 Dominant Share 的值实现对 Client 排序。DRFSorter 中的几个概念介绍如下。

(1) Client：资源的使用者。定义如下：

```
struct Client
{
    std::string name;           //资源使用者名称
    double share;               //资源使用者分配到的资源份额
};
```

Client 之间的排序规则是：首先比较 share 值的大小；如果 share 值相等，则比较 name 的大小。

(2) Resource：表示一种资源，包含资源 name、资源的值及资源被哪些 Framework 偏好。定义如下：

```
message Resource {
    required string name = 1;           //资源名称
    required Value.Type type = 2;        //资源值的类型
    optional Value.Scalar scalar = 3;    //值类型为标量，CPU 和内存的值类型都为标量
    optional Value.Ranges ranges = 4;    //值类型为范围
    optional Value.Set set = 5;          //值类型为集合
    optional string role = 6 [default = "*"]; //框架对于资源的偏好
}
```

Mesos 的应用环境是多资源类型的集群环境，因此框架可以申请混合资源，通过 Resource 数组来表示。

(3) Weight: 各个 Client 的资源分配的权重。定义的类型为 double。

为了对 Client 进行排序, DRFSorter 需要在内部维护如下一些信息。

- Dirty: 表示 DRFSorter 的资源分配发生改变, 需要重新计算各个 Client 的 share 值。
- Clients: 资源使用者的集合, 使用红黑树, 按照 Client 的 Dominant Share 的值进行排序。
- Resources: 此 DRFSorter 拥有的总资源值。

整个 DRFSorter 的接口有两类: 第一类是增加、删除和修改 Client 和 Resource, 在资源发生变化时, DRFSorter 会将 Dirty 置为 True; 第二类是对 Client 进行排序, 在 Dirty 为 true 的情况下, 会对所有的 Client 重新计算 Dominant Share 的值, 然后重新插入集合中, 返回排序结果。

DRFSorter 的核心排序流程如下:

```
| 如果 Dirty 标志位为 true
|   | 遍历 Clients 集合, 计算各个 Client 的 share 值
|   |   | 计算 Client 的各个 Resource 的值占 DRFSorter 此类型资源总值的比例,
|   |   |   找出最大值然后除以 Client 的权重, 就得到了 Client 的 share 值
|   |   |   公式为:  $Share = \max\{\sum R_i / T_i\} / w$ ,  $R_i$  为 client 分配到的资源  $i$  的值,
|   |   |    $T_i$  为资源  $i$  在 DRFSorter 中的总值,  $w$  为 Client 的权重
|   |   | 重新插入 Client, 进行重排序
| 返回排序后的结果
```

4) Allocator

Allocator 实现了一个分层的分配器, 用户可以根据 Role 对 Framework 进行分组, 在 RoleInfo 中指定 Framework Group 的权重, 专门有一个 DRFSorter 对各个 Framework 分组的资源分配进行排序。在各个 Framework Group 内部还有一个 DRFSorter 对 Group 内部的各个 Framework 的资源分配进行排序。

Allocator 中的一些概念如下。

(1) Framework: 应用程序框架, 向 Mesos 获取集群资源, 下发具体的计算任务, 是集群的使用者。

```
struct Framework
{
    hashset<Filter*> filters; //Framework 的过滤器
    bool checkpoint;         //是否正在进行 Checkpoint
    FrameworkInfo info;      //Framework 的信息
};
```

(2) Slave: 运行在各个集群节点的后台任务, 执行具体的计算任务, 上报节点的资源 and 负载。

```
struct Slave
{
    ...
    Resources available; //当前可用的资源
    bool whitelisted;    //是否在白名单中, 如果为 false, 则不能进行 Resource Offer
    bool checkpoint;    //是否正在进行 Checkpoint, 如果是, 则不能进行 Resource Offer
    SlaveInfo info;     //Slave 信息
};
```

```
};
```

(3) Role: 用于对 Framework 进行分组, 可以为每组 Framework 指定一个 Weight。Role 的信息 RoleInfo 如下:

```
message RoleInfo {
    required string name = 1;
    optional double weight = 2 [default = 1];
}
```

(4) Whitelist: 指定了有效的 Slave, 如果指定了白名单, 那么白名单内的 Slave 是有效的。

(5) 过滤器: 用于框架对 Slave 的资源进行过滤, 可以用于拒绝特定 Slave 上的资源。为了实现资源分配, Allocator 需要在内部维护以下信息。

- Frameworks: 框架的映射, 从 FrameworkId 到框架信息的映射。
- Slaves: Slave 的映射, 从 SlaveId 到 Slave 信息的映射。
- Roles: 框架分组信息的映射。
- Framework Sorters: 框架群组内部 DRF 排序容器的映射。
- Role Sorters: 框架群组之间进行 DRF 排序的容器。

Allocator 核心的调度逻辑如下:

```
| 对 Framework Group 进行从小到大排序 (Framework 按照 Role 进行分组)
| 遍历排序后的 Framework Group
|   | 在 Group 组内对各个 Framework 进行排序
|   | 遍历排序后的 Framework
|       | 遍历 Slave 集合, 将满足要求的所有 Slave 的可用资源发送给框架
|           | 提取 Slave 中 Role 为 "*" 的所有可用资源, "*" 表示一般普适的资源
|           | 提取与 Framework Group 的 Role 相同的所有可用的 Slave 资源
|               (此为提取框架偏好的 Slave 资源)
|           | 判断 Slave 资源是否满足条件:
|               | 过滤资源, 如果 Slave 或者 Framework 正在进行 Checkpoint 或者
|                   资源被 Framework 中设定的过滤器过滤掉, 则放弃资源
|               | Slave 不在白名单中, 放弃资源
|           | Slave 的资源小于最小资源限定, 放弃资源
|           | 将以上条件都满足的资源加入资源结果集合中,
|               并更新 Slave 的可用资源 (需要减去已经放入结果集合中的资源)
|           | 如果资源结果集合不为空
|               | 更新资源, 即 Slave 中的可用资源
|               | 执行 Resource Offer 操作, 将资源下发给框架
```

分配器会在以下几种情况下进行 Resource Offer:

- 新的 Slave 加入 Mesos 集群中。
- 新的 Framework 加入 Mesos 集群中。
- 定时地执行资源分配, 可以通过配置文件进行时间设置。

Allocator 在进行资源分配时, 提供了如下 4 个机制。

(1) 过滤器: 框架可以通过设置过滤器来拒绝特定资源。比如, Framework 在某个 Slave 上多次

执行失败，那么 Framework 就可以通过对这个 Slave 设定过滤器来拒绝这个 Slave 上的资源。

(2) Slave 的白名单：不在白名单内的 Slave 不参与 Resource Offer。

(3) Slave 的最小资源限定：为了适应请求大资源任务的框架，Mesos 通过设定 Slave 节点最小资源限定来避免在 Slave 上进行 Resource Offer，直到 Slave 上的空闲资源达到最小资源供给大小。

(4) 框架对 Slave 资源的偏好：通过 Resource 中的 Role 参数来实现，可以通过设定 Slave 上的资源的 Role 值及将框架按照 Role 进行分组来实现框架群组的专用资源和资源偏好。

2. YARN 调度算法

YARN 实现了 FIFO Scheduler、Capacity Scheduler 和 Fair Scheduler。第一个是默认的调度器，它属于批处理调度器。而后两个属于多租户调度器，采用树形多队列的形式组织资源。它们的核心资源分配模型是一样的。三种调度器解决的问题是：如何选择一个队列？在一个队列上如何选择一个应用？

YARN 的调度器进行资源分配的过程是：Scheduler 维护一群队列的信息，用户可以向一个或者多个队列提交应用。每次 NM 心跳的时候，调度器根据一定的规则选择一个队列，再在该队列上选择一个应用，然后尝试在这个应用上分配资源。

1) FIFO Scheduler

FIFO 是最简单的调度器，它按照先进先出的方式处理应用。只要有一个队列可以提交应用，那么所有用户都将应用提交到这个队列。如图 9.4 所示，FIFO 有一个队列，提交到队列的 job 1 和 job 2 按照 job 提交的先后顺序占用资源。

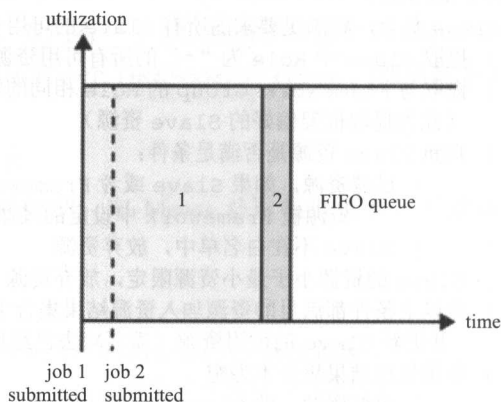


图 9.4

FIFO Scheduler 的不足：耗时的长任务会导致后提交的任务一直处于等待状态，如果这个集群是由多人共享的，显然不太合理。

2) Capacity Scheduler

Capacity Scheduler(容量调度器)是 Yahoo! 开发的多用户调度器，其最初的设计目的是使 Hadoop 应用能够被多用户使用，且最大化整个集群资源的吞吐量。它以队列为单位划分资源，每个队列可设定资源最低保证和使用上限，每个队列占用的集群资源是固定的，但是可以不同，队列内部仍然

采用 FIFO 调度策略。当队列的资源有剩余时,可暂时将剩余资源共享。如图 9.5 所示是 Capacity Scheduler 执行过程的示意图。

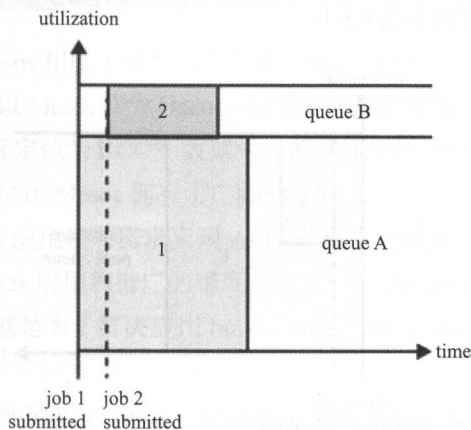


图 9.5

从图 9.5 中可以看到,队列 A 和队列 B 享有独立的资源,但是队列 A 所占的资源比重更大。如果在执行任务的时候,集群恰好有空闲资源,如队列 B 为空,那么调度器就可能分配更多的资源给队列 A,以更好地利用空闲资源。这种处理方式叫作 Queue Elasticity (弹性队列)。弹性队列也有一些副作用。如果此时队列 B 有了新任务,之前被队列 A 占用的资源并不会立即释放,只能等到队列 A 的任务执行完毕。为了防止某个队列过多地占用集群资源,YARN 提供了一项设置,可以控制某个队列能够占用的最大资源。但这其实又与弹性队列相冲突,因此这里有一个权衡的问题。

Capacity Scheduler 采用了三级资源分配策略(将双层调度机制中的第一层分为选择队列与选择应用程序两级),当一个节点上有空闲资源时,它会依次选择队列、应用程序和 Container (请求)使用该资源。

第一级:选择队列。不同队列之间按照队列的资源使用比排序。Capacity Scheduler 采用基于优先级的深度优先遍历算法选择队列:从根队列开始,按照资源使用率由小到大遍历子队列。如果子队列是叶子队列,则按照第二级、第三级的方法在队列中选择一个 Container;否则以该队列为根队列,重复以上过程,直到找到合适的 Container 并退出。

第二级:选择应用程序。在第一级选中一个叶子队列后,Capacity Scheduler 按照 Application ID 对叶子队列中的应用程序进行排序(先进先出),依次遍历排序后的应用程序,找到合适的应用程序。

第三级:选择 Container。选中一个应用程序后,先满足优先级高的 Container。同一优先级,先满足本地化的 Container。然后依次选择节点本地化、机架本地化和非本地化的 Container。

Capacity Scheduler 的特征:容量保证、灵活性、多重租赁、安全保证和动态更新配置文件等。

3) Fair Scheduler

Fair Scheduler (公平调度器)是 Facebook 开发的多用户调度器,与 Capacity Scheduler 有很多相似之处。二者的不同主要体现在资源公平共享、负载均衡、调度策略配置灵活和提高小应用的响应

时间等方面。Fair Scheduler 试图为每个任务均匀分配资源，比如，当前只有任务 1 在执行，那么它将拥有整个集群资源；此时任务 2 被提交，那么任务 1 和任务 2 将平分集群资源，以此类推。

Fair Scheduler 的执行过程如图 9.6 所示。

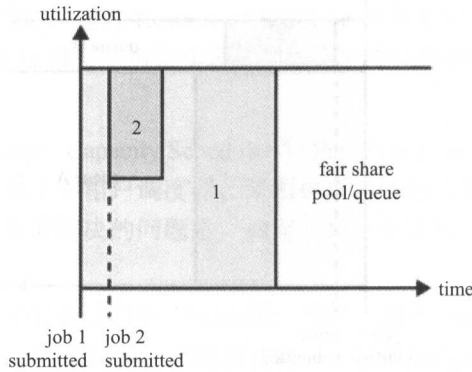


图 9.6

从图 9.6 中可以看出，队列 A 首先执行任务，任务 1 拥有整个集群资源；随后队列 B 增加任务 2，这两个队列均匀分配资源。

Fair Scheduler 也采用了三级资源分配策略，包括选择队列、选择应用程序和选择满足应用程序的资源。与 Capacity Scheduler 不同的是，Fair Scheduler 提供了更多样化的调度策略，调度策略在队列间和队列内部可单独设置。当前实现的策略包括如下三种。

(1) 先来先服务 (FIFO)：按优先级高低调度；优先级相同，则按提交时间先后顺序调度；提交时间相同，则按名称大小调度。

(2) 公平调度 (Fair)：按内存资源比率大小调度。

(3) 主资源公平调度 (Dominant Resource Fairness)：按主资源公平调度算法调度，所需份额最大的资源称为主资源。把最大最小公平算法应用于主资源上，将多维资源调度问题转换为单资源调度问题。

三种资源调度器的对比如表 9.1 所示。

表 9.1

	FIFO Scheduler	Capacity Scheduler	Fair Scheduler
设计目的	最简单的调度器，易于理解和上手	在多用户的情况下，最大化集群的吞吐和利用率	在多用户的情况下，强调用户公平地贡献资源
队列组织方式	单队列	树状组织队列。无论是父队列还是子队列，都会有资源参数限制，子队列的资源限制计算是基于父队列的。应用提交到叶子队列	树状组织队列。但是父队列和子队列没有参数继承关系，父队列的资源限制对子队列没有影响。应用提交到叶子队列
队列排序算法		按照队列的资源使用量最小的优先	根据公平排序算法排序
应用选择算法	先进先出	先进先出	先进先出或者公平排序算法

9.4 数据中心统一资源调度

在传统的 Hadoop 领域, Mesos 相比 YARN 没有太多优势, YARN 一出现就是 Hadoop 的原生软件, 生态的优势要远大于 Mesos, 所以 Mesos 作为 Hadoop 领域的资源管理软件的目标可谓步步维艰。

Mesosphere 将 Mesos 的目标定位为 DCOS(数据中心的操作系统)非常明智, 期望打造类似 Google Borg 的数据调度中心, 这一定位让 Mesos 拥有了广阔的空间。

Mesos 原生的调度接口需要应用按照需求实现 API, 这种机制需要应用全部重写, 这种私有标准很难为大家接受, 从而使得 Mesos 应用和推广的难度非常大, 而 Docker 恰好完美地解决了这个问题。Docker 天生就是应用的容器, 就是为了解决应用 build、ship、run 的问题, Mesos 在不知不觉中变成了 PaaS 调度器。

Mesos 对 PaaS 的支持有多种方式, 如 Mesos+Marathon 或者 Mesos+Kubernetes。前者是 Mesosphere 公司推出的 PaaS 服务, 后者是 Google 的 Borg 的简化版本。通过 Mesos+Marathon 或者 Mesos+Kubernetes, 使得大数据和传统常驻应用也可以一起部署到一个集群, 实现整个数据中心的调度。

9.4.1 Mesos+Marathon 架构和原理

1. Marathon 概览

Marathon 本质上是 Mesos 实现的一个 Framework, 能够支持运行长服务, 如 Web 应用等; 类似集群的分布式 Init.d, 能够原样运行任何 Linux 二进制发布版本, 如 Tomcat Play 等, 可以进行集群的多进程管理; 也是一种私有的 PaaS, 为部署提供 REST API 服务, 有授权和 SSL、配置约束, 通过 HAProxy 实现服务发现和负载均衡, 可以理解为一个简化的 PaaS 平台。

通过 Marathon, 我们可以如同管理一台 Linux 主机一样管理数千台服务器。它们的对应原理如图 9.7 所示, Marathon 的作用类似于 Linux 主机内的 init Systemd, 而 Mesos 则进行复杂的资源管理, 把 4 台服务器当成一个 Linux 系统。

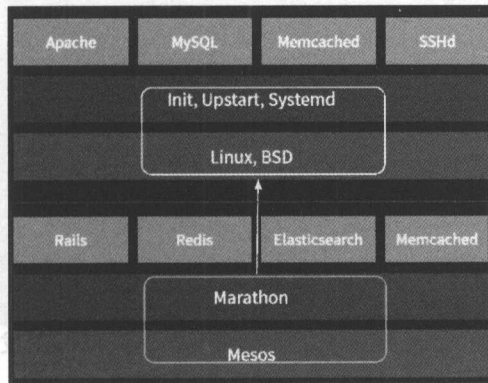


图 9.7

Marathon 提供了一组 REST API，在不同粒度上管理应用、组、任务、事件等。对于每个 API，都接受一段预定格式的 JSON 来指定参数。同时，Marathon 提供了简单易用的 Web UI，可以创建简单的应用及观察应用结果，如图 9.8 和图 9.9 所示。

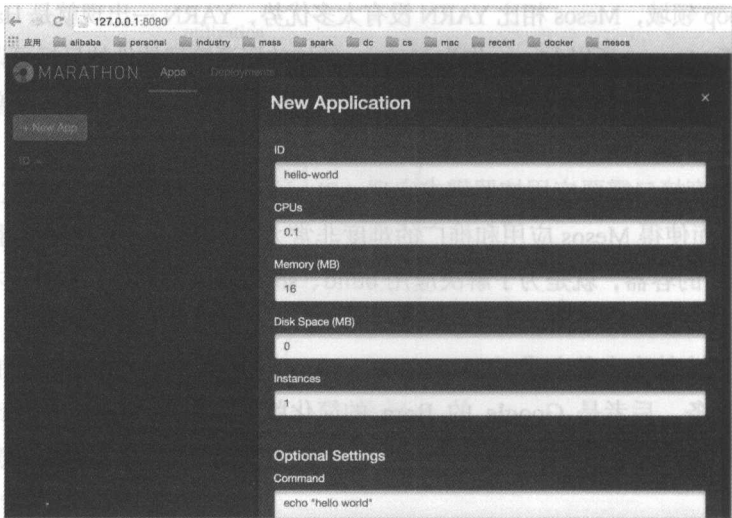


图 9.8

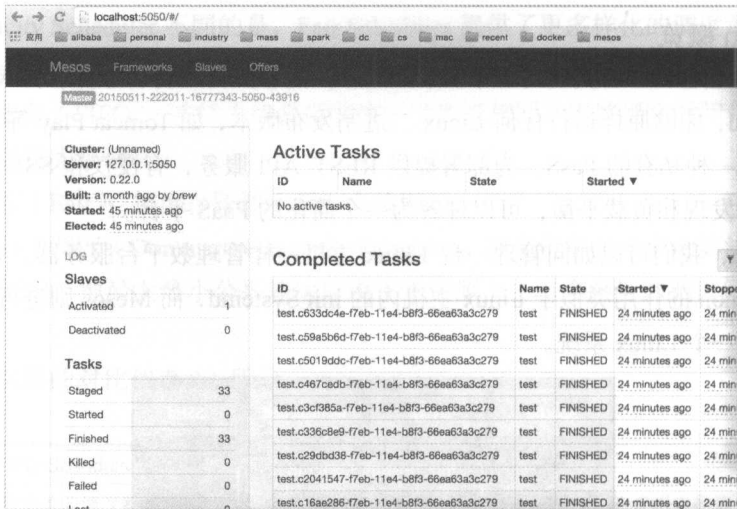


图 9.9

2. Marathon 底层框架

Marathon 是由 Scala 语言编写的，运行在 Mesos 之上的一个 Framework，提供了应用部署、管理和监控等功能，实现了一个简易的 PaaS 系统。

Marathon 采用 Akka 实现的 Actor 异步编程模型具有极其高效的并发处理能力，同时利用了 Chaos 提供的 IOC 框架、Web 服务和 API 发布功能，如图 9.10 所示。

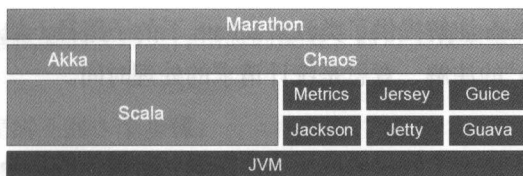


图 9.10

1) Chaos

Chaos 是 Mesosphere 的一个子项目，目前应用在 Marathon 和 Chronos 上，是一个专门为 Scala 编写的轻量级的 REST 服务发布框架。它的设计初衷是希望实现一个轻量级、简单易用、采用 Scala 语言、专门做 REST 服务发布的框架。它运行在 JVM 之上，实际上集成了如下几个框架和库。

- Jersey: Oracle 推出的 RESTful 框架。
- Guice: 一个轻量级的依赖注入框架。
- Guava: 一个提供了生命周期管理和大量工具类的 Java 库。
- Jetty: Web 服务器和 Servlet 容器。
- Jackson: JSON 工具库。
- Hibernate Validator: 提供 API 参数校验功能。
- Coda Hale's Metrics: 提供 JVM 和应用的性能数据统计。

下面简单介绍一下 Jersey 和 Guice 这两个核心的组件。

① Jersey

JAX-RS 是用于 RESTful Web Services 的 Java API。Jersey 是官方提供的一个参考实现。它提供了一套标准化的注解和 API，方便以非常直观的方式开发 RESTful Web Services。其编码风格如图 9.11 所示。

```

@Path("foo")
@Produces(Array(MediaType.APPLICATION_JSON))
class ExampleResource {

    private final val names = Seq("Dude", "Walter", "Donny", "Jesus")

    @GET
    @Timed
    def get() = {
        val person = new Person
        person.name = names(Random.nextInt(names.size))
        person.age = Random.nextInt(150)
        person
    }

    @POST
    @Timed
    def post(@Valid person: Person) {
        println(person)
    }

    @Path("bar")
    def bar() = new ExampleSubResource
}
  
```

图 9.11

@Path 注解表明 REST 资源访问 URL 路径，函数上的 @GET、@POST 等注解表明访问 HTTP

Method。在 bar 函数上的 @Path 注解提供了类注解 @Path 下的子路径，如 /foo/bar 即可访问 bar 函数，Timed 是 Coda Hale's Metrics^① 的注解，意思是统计请求的处理时间。

② Guice

Guice 是类似 Spring 框架的一个轻量级应用程序框架，它提供的核心功能是 IOC 和 AOP。

IOC（控制反转）是 Guice 的核心功能，AOP 也是依赖于它实现的。那么，什么是控制反转呢？传统的程序代码直接负责对象的创建，如图 9.12 所示。

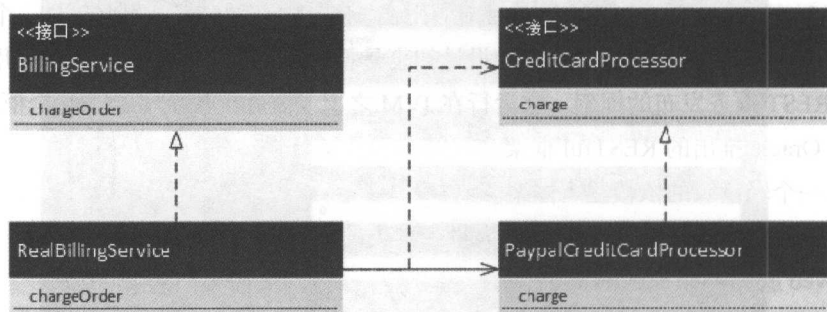


图 9.12

RealBillingService 类负责创建 PaypalCreditCardProcessor 和 DatabaseTransactionalLog 两个实现类，也就是说，订单服务的代码直接依赖于信用卡处理和事务日志的实现类。这种实现之间的强耦合带来的问题是，CreditCardProcessor 和 TransactionLog 接口的不同实现的转换都需要直接修改 RealBillingService 的源代码，严重影响了程序的测试和模块之间的解耦。

解决方法是各模块代码不依赖其他模块的实现，只依赖接口，如图 9.13 所示。

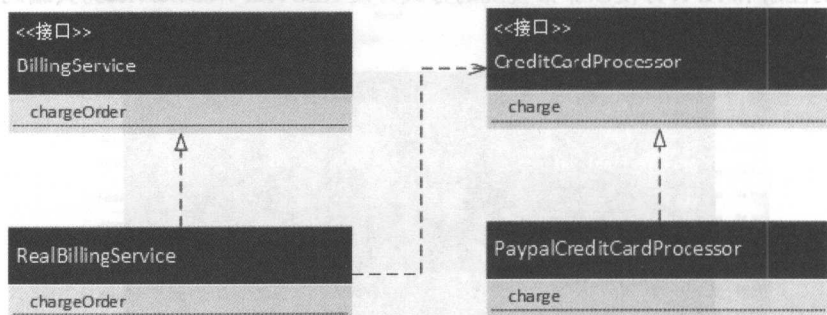


图 9.13

图 9.13 所示的程序不再直接依赖 CreditCardProcessor 和 TransactionLog 接口的实现，只能由调用它的代码通过构造函数注入它的对象中。

这就是 IOC 的一种实现：依赖注入。Guice 和大多数 IOC 框架都采用这种实现。框架在运行时创建一个 IOC 容器，业务代码只关注业务逻辑，不再关注对象的创建和组装。这一切都由容器来

^① 参考 <http://metrics.codahale.com>。

完成。应用开发者只需根据框架的要求进行相应的配置，即可实现对象注入。Guice 配置采用注册的方式实现。

使用 Guice 框架需要完成下面 3 个步骤：

- 在需要代码注入的地方标注 @Inject 注解。
- 创建 Module 类，定义注入规则。
- 创建并使用容器。

其中根据注入的方式不同，@Inject 可以定义在代码的不同位置。分为：

- 构造方法注入。
- 方法参数注入。
- 属性注入。

2) Akka

要想编写出正确的、具有容错性和可扩展性的并发程序实在太困难了，这多数是因为使用了错误的工具和抽象级别。Akka 就是为了改变这种状况而生的。通过使用 Actor 模型提升了抽象级别，为构建正确的可扩展并发应用提供了一个更好的平台。Actor 还为透明地构建分布式高容错、高扩展系统打下了基础。

① Actor

相对于线程来说，Actor 是一种更高层次抽象的并发编程模型。它屏蔽了线程创建、销毁、锁竞争等物理细节。

Actors 可以提供：

- 对并发/并程序的简单的、高级别的抽象。
- 异步、非阻塞、高性能的事件驱动编程模型。
- 非常轻量的事件驱动处理（1GB 内存可容纳约 270 万个 Actors）。

一个 Actor 是一个容器，它包含了状态、行为、一个邮箱、子 Actor 和一个监管策略。所有这些都包含在一个 Actor 引用里。Actor 引用是外界访问 Actor 的唯一方式。外界和 Actor 的互动都是通过操作 Actor 引用来发送消息的方式实现的。Actor 之间没有共享状态，所以不需要锁来实现资源同步，可以完全实现无锁设计。

像一个经济学组织一样，Actor 自然会形成树形结构。程序中负责某一个功能的 Actor 可能需要把它的任务分拆成更小的、更易管理的部分。为此，它启动子 Actor 并监管它们。唯一的前提是要知道每个 Actor 有且仅有一个监管者，也就是创建它的那个 Actor。

② Actor 的消息传递

通过以下代码片段来直观感受一下 Actor 的创建和消息传递过程。

```
object StudentSimulatorApp extends App{
  //Initialize the ActorSystem
  val actorSystem=ActorSystem("UniversityMessageSystem")
  //construct the Teacher Actor Ref
```

```

val teacherActorRef=actorSystem.actorOf(Props[TeacherActor])
//send a message to the Teacher Actor
teacherActorRef ! QuoteRequest
//Let's wait for a couple of seconds before we shut down the system
Thread.sleep (2000)
//Shut down the ActorSystem.
actorSystem.shutdown()
}

class TeacherActor extends Actor {
  val quotes = List(
    "Moderation is for cowards",
    "Anything worth doing is worth overdoing",
    "The trouble is you think you have time",
    "You never gonna know if you never even try")
  def receive = {
    case QuoteRequest => {
      import util.Random
      //Get a random Quote from the list and construct a response
      val quoteResponse=QuoteResponse(quotes (Random.nextInt (quotes.size)))
      println (quoteResponse)
    }
  }
}

```

上述代码首先创建了一个 TeacherActor，随后通过 teacherActorRef 这个 Actor 引用，向已创建的 Actor 发送了一个消息。Thread.sleep(2000)是在等 TeacherActor 处理完这个消息。TeacherActor 在 receive()方法中定义了消息处理方式。实际运行过程如图 9.14 所示。

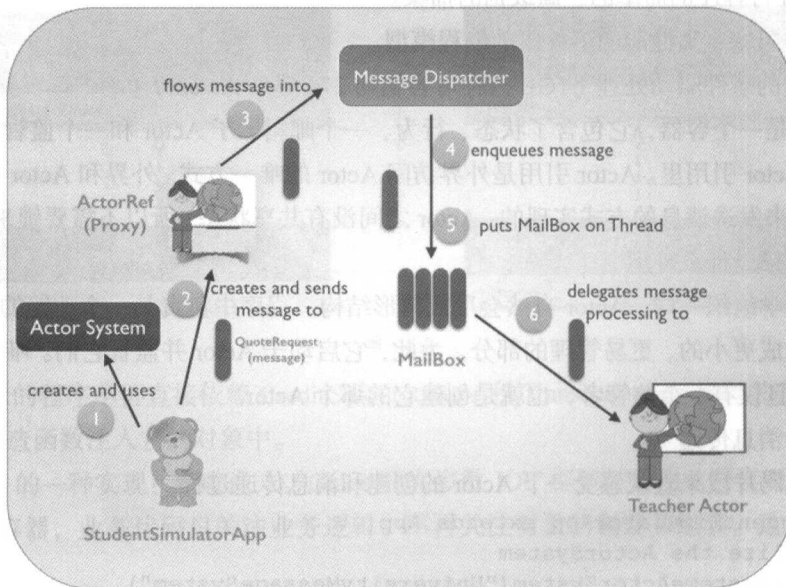


图 9.14

主程序首先创建了 ActorSystem 实例, ActorSystem 是 Actor 世界的入口。由于 Actor 是树形结构, 因而 ActorSystem 也可以看作树的根, 是其他所有 Actor 的父 Actor。

然后创建了一个 TeacherActor 的 Actor 实例, 并获得该 Actor 的引用类实例 teacherActorRef。teacherActorRef 并不是 Actor 的对象, 而是 ActorRef 的对象。这里就严格限制了代码不会直接操作 Actor, 不会直接获取 Actor 的状态。对 Actor 的操作都是通过向 ActorRef 发送消息完成的。teacherActorRef 可以看成 TeacherActor 实例的一个代理。

接着, 主程序向 teacherActorRef 发送一个 QuoteRequest 消息。

随后, teacherActorRef 将消息的处理发送到 Message Dispatcher。Message Dispatcher 可以说是 Actor 消息系统的核心和引擎。

Message Dispatcher 会将消息加入 TeacherActor 的 Mailbox 中。实际上, 由于 Dispatcher 是整个系统的引擎, 因此, 它还负责消息处理过程的发送。这里的做法是: 执行 Mailbox。Mailbox 实际上是一个实现了 Runnable 接口的类, 可以放到线程池中运行。

Mailbox 运行后, 会将消息出列, 并调用 Actor 的 receive 函数进行处理。

③ Actor 生命周期

Actor 的整个生命周期如图 9.15 所示。

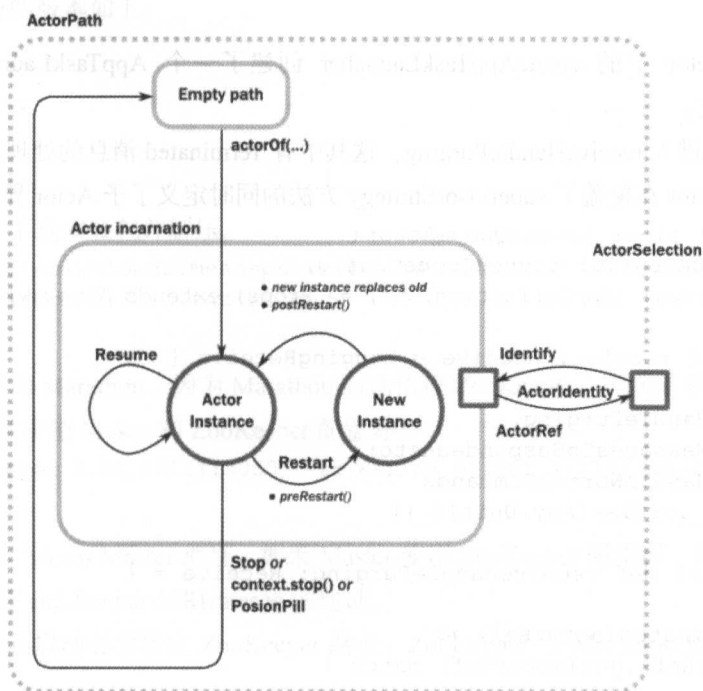


图 9.15

Akka Actor 中定义了下列回调钩子。

- **preStart**: 在 Actor 实例化后执行, 重启时不会执行。

- `postStop`: 在 Actor 正常终止后执行，异常重启时不会执行。
- `preRestart`: 在 Actor 异常重启前保存当前状态。
- `postRestart`: 在 Actor 异常重启后恢复重启前保存的状态。

可以调用 `ActorSystem` 的 `actorOf` 函数创建一个 Actor。在 Actor 的代码内部也可以通过 `context.actorOf` 创建子 Actor。同时还可以创建 Actor 的多个实例，通过 `Props.withRouter` 定义负载均衡器来调度消息。

Actor 的销毁也有多种方式，可以由父 Actor 调用 `context.stop` 函数停止，或由自己调用 `context.stop(self)` 来停止。也可以给 Actor 发送 `PoisonPill` 或 `Kill` 消息来停止。这两个消息的区别是，前者停止前会发送一个 `Terminated` 消息给所有的 `Watchers`；而后者会抛出一个 `ActorKilledException` 异常并传送到它的 `Supervisor`，也就是父 Actor。

④ 子 Actor

在 Actor 代码中通过 `context.actorOf` 创建的 Actor 是该 Actor 的子 Actor。父 Actor 可以通过 `context.watch` 函数监控子 Actor，也可以覆盖 `supervisorStrategy` 来定义子 Actor 异常的处理方式。

以下是 Marathon 使用任务队列相关代码，`LaunchQueueActor` 类管理了一个任务队列，它会对不同的应用程序创建 `AppTaskLauncherActor`，`AppTaskLauncherActor` 负责该应用程序的所有任务的启动。

`LaunchQueueActor` 中的 `createAppTaskLauncher` 创建了一个 `AppTaskLauncherActor`，并通过 `context.watch` 监控它。

`receive()` 方法会进入 `receiveHandlePurging`，这其中有 `Terminated` 消息的处理逻辑。

`LaunchQueueActor` 在覆盖了 `supervisorStrategy` 方法的同时定义了子 Actor 异常的处理方式。

```
private[impl] class LaunchQueueActor(
  launchQueueConfig: LaunchQueueConfig,
  appActorProps: (AppDefinition, Int) => Props) extends Actor with ActorLogging {
  ...
  override def receive: Receive = LoggingReceive {
    Seq(
      receiveHandlePurging,
      receiveMessagesToSuspendedActor,
      receiveHandleNormalCommands
    ).reduce(_._2.orElse[Any, Unit](_))
  }
  private[this] def receiveHandlePurging: Receive = {
    ...
    case Terminated(actorRef) =>
      launcherRefs.get(actorRef) match {
        ...
      }
  }
  ...
  private[this] def createAppTaskLauncher(app: AppDefinition, initialCount: Int):
```



```
ActorRef = {
  val actorRef = context.actorOf(appActorProps(app, initialCount), s"$childSerial-
${app.id.safePath}")
  ...
  context.watch(actorRef)
  actorRef
}
override def supervisorStrategy: SupervisorStrategy = OneForOneStrategy() {
  case NonFatal(e) =>
    // We periodically check if scaling is needed, so we should recover. TODO:
Speedup
    // Just restarting an AppTaskLauncherActor will potentially lead to starting
too many tasks.
    Stop
  case m: Any => SupervisorStrategy.defaultDecider(m)
}
```

3. Marathon 的安装和使用

1) 安装和启动

(1) 安装依赖。

依赖软件和对应的版本如下：

Apache Mesos 0.22.1

Apache ZooKeeper

JDK 1.7+

(2) 安装 Marathon。

执行如下命令，下载安装包并解压。

```
$ curl -O http://downloads.mesosphere.com/marathon/v0.10.1/marathon-0.10.1.tgz
$ tar xzf marathon-0.10.1.tgz
```

(3) 启动。

执行如下命令启动 Marathon，因为 Marathon 启动依赖 ZooKeeper，因而需要首先启动 Mesos 和 ZooKeeper，并在参数中把 Mesos 和 ZooKeeper 配置好。

```
$ ./start --master 9.91.201.1:5050 --zk zk://9.91.201.1:2181/marathon-performance
```

参数解析如下。

- **--master**: 指定 Mesos Master 地址，如果 Master 使用 ZooKeeper 做集群，则指定类似 zk://zk1.foo.bar:2181 或 zk2.foo.bar:2181/mesos 的地址。
- **--zk**: Marathon 启动必须指定 ZooKeeper 路径。ZooKeeper 为 Marathon 提供主备选择和状态的存储。

(4) 创建应用。

- 通过页面创建应用，如图 9.16 所示。

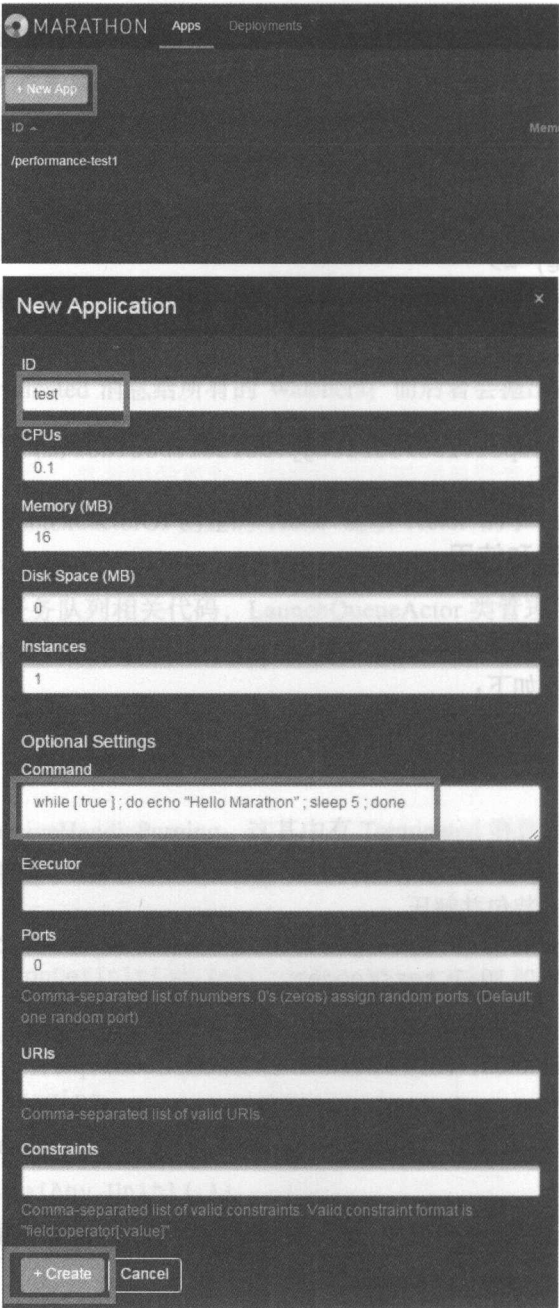


图 9.16

b. 通过 API 创建应用，代码如下：

```
curl -X POST -H "Content-Type:application/json" http://9.91.34.102:8080/v2/apps -d '{
  "id": "simple-app-rest",
```

```
"cmd": "while [ true ] ; do echo 'Hello Marathon' ; sleep 5 ; done",
"cpus": 0.1,
"mem": 10.0,
"instances": 1
}'
```

2) 基本特性

① 资源下载

请求在 body 中添加 `uris` 配置项, 即可下载外部的源码、可执行文件等到 Slave 上运行。

```
{
  "id": "app-with-external-res",
  "cmd": "chmod u+x ./test.sh; ./test.sh",
  "cpus": 0.1,
  "mem": 10.0,
  "instances": 1,
  "uris": [
    "http://rnd-github.huawei.com/f00231050/gittest/raw/master/test.zip"
  ]
}
```

② Artifact Store

Marathon 会在本地存储外网文件, 在多次部署任务时, 第一次部署 Marathon 可将外部下载的资源文件存储到本地, 后续部署 Slave 不再从外网下载资源, 而是从 Marathon 提供的 Artifact 资源服务器下载。

```
{
  "id": "app-with-artiface-res",
  "cmd": "chmod u+x ./test.sh; ./test.sh",
  "cpus": 0.1,
  "mem": 10.0,
  "instances": 2,
  "storeUrls": [
    "http://rnd-github.huawei.com/f00231050/gittest/raw/master/test.zip"
  ]
}
```

Marathon 启动参数中有 `--artifact_store` 文件存储路径。

③ Docker 容器应用

Marathon 支持部署 Docker 容器; 支持文件系统映射和网络桥接。

```
{
  "id": "python-web",
  "cmd": "python3 -m http.server 8080",
  "cpus": 0.5,
  "mem": 32.0,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "python:3",

```

```

        "network": "BRIDGE",
        "portMappings": [
            {
                "containerPort": 8080,
                "hostPort": 0
            }
        ]
    }
}

```

④ 健康度检查

Marathon 支持配置应用层的健康度检查，支持 HTTP、TCP 协议和 Command 方式。配置样例如下：

```

{
    "id": "python-web",
    "cmd": "python3 -m http.server 8080",
    "cpus": 0.5,
    "mem": 32.0,
    "container": {
        "type": "DOCKER",
        "docker": {
            "image": "python:3",
            "network": "BRIDGE",
            "portMappings": [
                {
                    "containerPort": 8080,
                    "hostPort": 0
                }
            ]
        }
    }
    "healthChecks": [
        {
            "protocol": "HTTP",
            "path": "/",
            "gracePeriodSeconds": 3,
            "intervalSeconds": 10,
            "portIndex": 0,
            "timeoutSeconds": 10,
            "maxConsecutiveFailures": 3
        }
    ]
}

```

⑤ 滚动升级

Marathon 更新应用时，支持滚动升级，即先停止一部分老版本的实例，然后启动一部分新实例

开始分流老版本流量，最终逐步替换成新版本。注意，要保证升级过程中的业务不间断。在应用配置中添加 `upgradeStrategy` 域和 `minimumHealthCapacity` 表示：

```
{
  "id": "simple-app-rest",
  "cmd": "while [ true ] ; do echo 'Hello Marathon' ; sleep 5 ; done",
  "cpus": 0.1,
  "mem": 10.0,
  "instances": 1
  "upgradeStrategy": {
    "minimumHealthCapacity": 0.5,
    "maximumOverCapacity": 0.2
  }
}
```

其中，`minimumHealthCapacity` 为升级过程中保证的最小可用节点数，`maximumOverCapacity` 为
保证升级过程中节点数不会超过当前节点的比例。

⑥ 约束条件

Marathon 支持配置应用部署的约束条件，可实现应用内的亲和性、反亲和性部署。

```
$ curl -X POST -H "Content-type: application/json" localhost:8080/v2/apps -d '{
  "id": "sleep-unique",
  "cmd": "sleep 60",
  "instances": 3,
  "constraints": \[\["hostname","UNIQUE"\]\]
}'
```

支持的操作符包括如下几个。

- **UNIQUE**：所有 Task 部署的 Slave 中的相应属性都不相同。在上例中，要保证所有 Task 部署到不同的主机上。
- **CLUSTER**：所有 Task 部署的 Slave 都有相同的属性。`"constraints": \[\["hostname","CLUSTER", "a.specific.node.com"\]\]` 配置所有 Task 都运行在 `a.specific.node.com` 这台主机上。
- **GROUP_BY**：将 Task 均匀分布到某些不同属性的 Slave 上。
- **LIKE**：属性模糊匹配。
- **UNLIKE**：与 LIKE 相反，部署在那些属性不匹配的 Slave 上。

9.4.2 Mesos+Marathon 小结

目前 Mesos、Kubernetes 等平台还在发展中，未来可能会发展成真正的数据库操作系统，对应的直接竞争对手应该是 OpenStack。这些 PaaS 平台从解决应用部署和快速开发出发，蚕食 IaaS 资源管理领域，应该说未来的应用前景可能会更大。

9.5 多租户技术

9.5.1 多租户概念

多租户技术 (Multi-Tenancy Technology) 又称多重租赁技术, 用于实现如何在多用户的环境下共用相同的系统或程序组件, 并且仍可确保各用户间数据的隔离性。

具体的多租户技术有多种, 数据库通常有以下三种。

1. 独立数据库

这是第一种方案, 即一个租户一个数据库。这种方案的用户数据隔离级别最高、安全性最好, 但成本也高。

优点: 为不同的租户提供独立的数据库, 有助于简化数据模型的扩展设计, 满足不同租户的独特需求; 如果出现故障, 则恢复数据比较简单。

缺点: 增大了数据库的安装数量, 随之带来维护成本和购置成本的增加。这种方案与传统的一个客户、一套数据、一套部署类似, 差别只在于软件统一部署在运营商那里。如果面对的是银行、医院等要求数据隔离级别非常高的租户, 则可以选择这种模式, 提高租用的定价。如果定价较低, 产品走低价路线, 那么这种方案对运营商来说是无法承受的。

2. 共享数据库, 隔离数据架构

这是第二种方案, 即多个或所有租户共享 Database, 但一个 Tenant 一个 Schema。

优点: 为安全性要求较高的租户提供了一定程度的逻辑数据隔离, 但并不是完全隔离; 每个数据库可以支持更多的租户数量。

缺点: 如果出现故障, 则数据恢复比较困难, 因为恢复数据库将涉及其他租户的数据; 如果需要跨租户统计数据, 则存在一定的困难。

3. 共享数据库, 共享数据架构

这是第三种方案, 即租户共享同一个 Database、同一个 Schema, 但在表中通过 Tenant ID 区分租户的数据。这是共享程度最高、隔离级别最低的模式。

优点: 三种方案比较, 第三种方案的维护和购置成本最低, 允许每个数据库支持的租户数量最多。

缺点: 隔离级别最低, 安全性最低, 需要在设计开发时加大对安全的开发量; 数据备份和恢复最困难, 需要逐表逐条备份和还原。如果希望以最少的服务器为最多的租户提供服务, 并且租户接受以牺牲隔离级别换取降低成本, 那么这种方案最合适。

9.5.2 多租户方案

在大数据技术里面, 实现多租户会有多种部署模式。与传统数据库不同的是, 在大数据场景下, 客户通常希望不同租户之间的数据可以共享, 但计算资源可以相互隔离而互不影响。

例如，一家企业有两个租户，一个租户做 ETL 计算，另一个租户做一些基础的分析。为了实现多租户，会有多种不同的部署方式。

方案 1：ETL 和基础分析合并部署为一个 Hadoop 集群，并为数据处理和数据分析分别设置不同的租户，通过对两类租户设置不同的资源上限，实现资源隔离，做到互不影响，如图 9.17 所示。

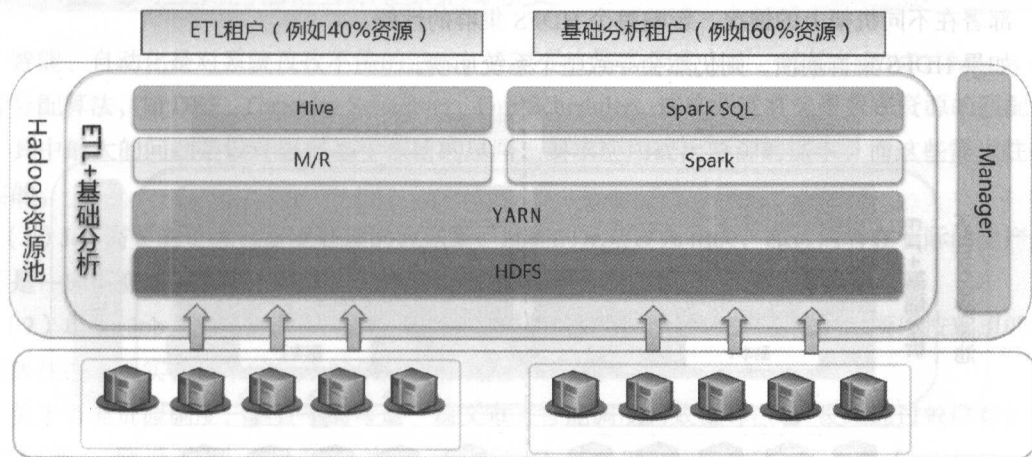


图 9.17

优点：

- 这是较常规的部署方式，案例很多，比较稳定。
- 部署简单，维护简单，通过 YARN 进行资源隔离简单、方便。
- 修改资源的配额简单。
- 集群的扩容简单，并且不需要更改资源配额。
- 资源利用率高，所有计算均可以利用所有节点的计算资源。

缺点：

- 共用 YARN，如果 YARN 崩溃，则 ETL 和 Hadoop 的计算都将崩溃。
- YARN 的隔离是逻辑隔离，不如物理隔离更彻底。

方案 2：ETL 和基础分析共享一个 HDFS，控制计算资源的 YARN 部署为两个，分别为数据处理和数据分析服务，做到计算存储资源的共享和计算资源的物理隔离。同时通过 Hadoop 的机架感知能力，保证三个副本的数据至少在 ETL 和基础分析计算集群所在的节点上各有一个副本，达到计算的本地化，如图 9.18 所示。

优点：

计算资源物理隔离，两个 YARN 没有任何影响，一个崩溃不会影响另一个。

缺点：

- 这种方案部署复杂、维护复杂，相当于维护两个集群。
- 修改配额等于重新配置 YARN 集群。

- 系统扩容后，两个计算集群均需要重新配置。
- 计算资源利用率低，可能出现一个计算集群繁忙、另一个计算集群闲置的情况。
- 让三个副本保证两个计算集群所在的节点各有一个，目前的社区版本无此方案，只能通过将两个集群设置为两个逻辑机架的方式实现。但是这样部署很可能会出现大量的三个副本分别部署在不同机架上的情况，影响整个 HDFS 集群的性能。
- 如果 HDFS 集群崩溃，则仍然会导致整个系统崩溃。

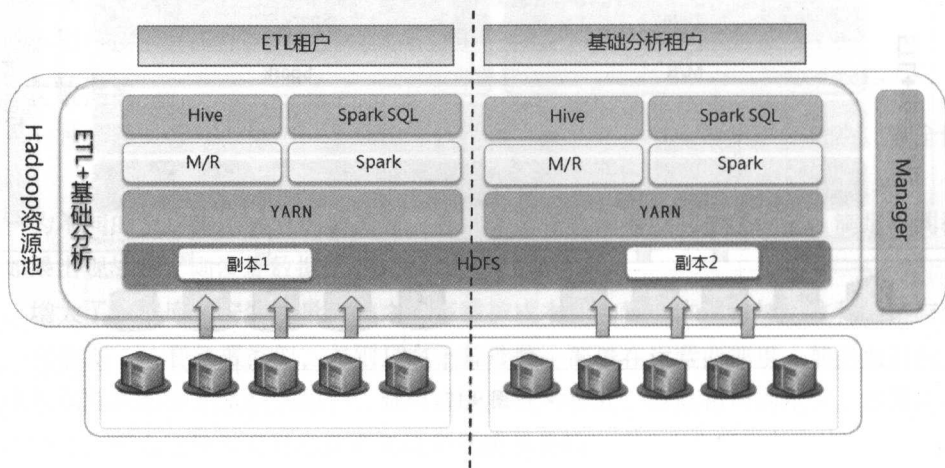


图 9.18

方案 3：ETL 和基础分析合并部署为一个 Hadoop 集群，并为数据处理和数据分析分别设置不同的 Hive、Spark 等组件实例。实例可以指定具体部署的物理机或者容器，通过实例做到物理隔离；在 YARN 之上的计算资源完全隔离，做到互不影响，如图 9.19 所示。

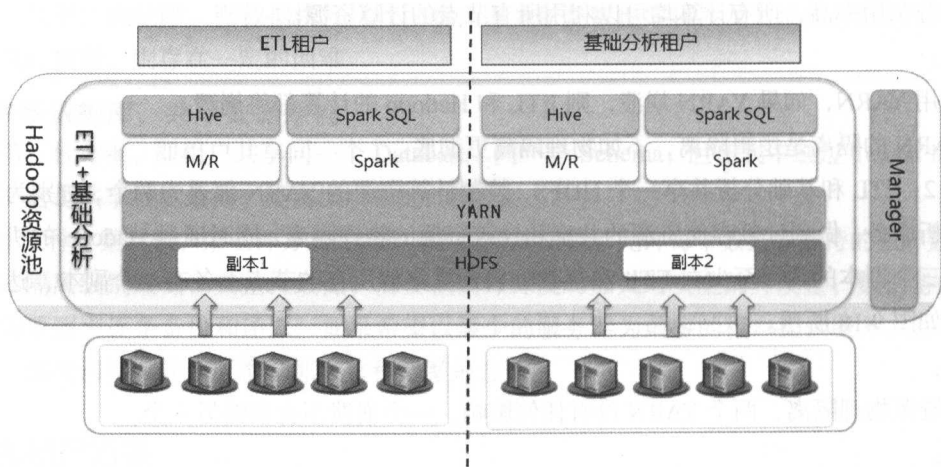


图 9.19

方案 3 是方案 1 的升级版，相比方案 1 有更好的隔离性。

三种方案都有优缺点，第三种方案相对来说更折中一些。在实际应用中，需要根据不同的场景采用合适的方案。

9.6 基于应用描述的智能调度

智能、自动化是对系统孜孜不倦的追求，尤其是在资源调度方面。传统的 YARN/Mesos 有各种资源分配算法，如 DRF、Capacity Scheduler、Fair Scheduler。这些调度算法聚焦在资源的匹配和分配上。其中最大的问题是设计思路基于预留的思路，要求应用提出资源的需求，而这些需求往往是不现实的。

(1) Job 需要的资源和需要处理的数据量、过程的复杂度强相关，而这两者在实际的生产系统中都不是一成不变的，所以往往预留的资源要么过大、要么过小。

(2) 单个 Job 本身也是一个复杂的过程，需要的资源也不是一成不变的，所以按理想的状态预留资源往往不现实，带来的最终结果是资源利用率不高。

关于智能资源调度，笔者早前写过一篇文章《智能调度与蚁群算法》，设想通过蚁群算法来自动分配资源。最近看到 Stanford 的一篇论文 *Quasar: Resource-Efficient and QoS-Aware Cluster Management*，通过简单的分类算法就达到了 Job 资源的预测和自动分配，解决了上面提到的两个资源分配难题。

Quasar 的基本思路如下。

(1) Quasar 不是基于资源预留和匹配的模式。Quasar 提供一个高层次的接口让用户为调度器描述其性能约束条件。不同类型的 workload，接口也不一样。比如，对于延迟苛刻的 workload，限制条件为 QPS（查询次数/秒）；分布式框架，如 Hadoop，限制条件是执行时间；对于单机里面的单线程或者多线程 workload，限制条件为低层次的 IPS（每秒执行的指令数）。

(2) Quasar 使用快速分类技术评估不同的资源分配导致的不同 workload 性能。一个新进的 workload 和数据集，需要从一些 Server 上采集一段时间（从几秒到几分钟）。这些有限的采集信息和已有的被分类离线的 workload、已经被调度过的 workload 组合起来。分类的结果准确地估计了应用的性能，考虑各种不同类型和数量的服务器及单服务的资源量，排除了其他一些干扰。Quasar 将 workload 资源分配问题简化成 4 种主要资源的分配，分配的服务器类型和干扰的程度。这大大降低了分类的复杂性问题。

(3) Quasar 将分类的结果直接用于资源的分配，消除了分配和匹配的二次无效性。Quasar 使用贪心算法结合 4 个主要资源的分类结果，综合选择数量或具体的资源集，来尽量满足业务提出的性能要求。Quasar 同时监视系统性能。如果在有空闲资源的情况下约束没有得到满足，即使 workload 发生了变化（如 Job 内部阶段发生了变化），则说明分类是不正确的，或者使用贪婪算法的次优解。因此，只要条件允许，Quasar 会重新分配资源。

9.7 Apache Mesos 架构和原理^①

Hadoop 领域的资源框架有很多，本节来分析 Apache Mesos 的架构和原理。

9.7.1 Apache Mesos 背景

Apache Mesos 是由加利福尼亚大学伯克利分校的 AMPLab 首先开发的一款开源群集管理软件，支持 Hadoop、ElasticSearch、Spark、Storm 和 Kafka 等架构，如图 9.20 所示。由于其开源性质，越来越受到一些大型云计算公司的青睐，如 Twitter、Facebook 等。Mesosphere 是围绕 Mesos 开展商业活动的初创公司。

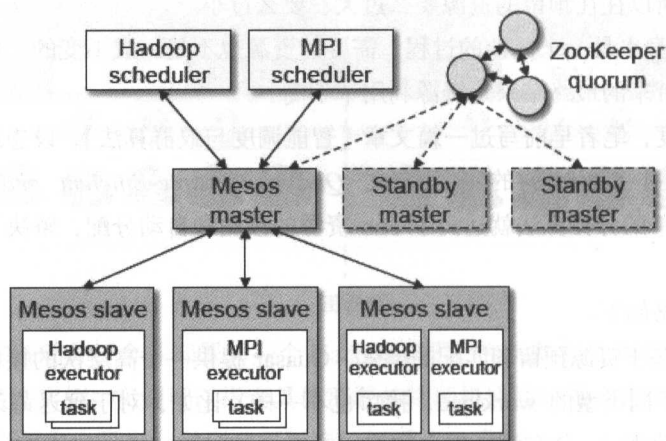


图 9.20

从总体上看，Mesos 是一个 Master/Slave 结构，其中，Master 是非常轻量级的，仅保存了 Framework（各种计算框架称为 Framework）和 Mesos Slave 的一些状态，而这些状态很容易通过 Framework 和 Slave 重新注册而重构，因而很容易使用 ZooKeeper 解决 Mesos Master 的单点故障问题。

Mesos Master 实际上是一个全局资源调度器，采用某种策略将某个 Slave 上的空闲资源分配给某个 Framework，各种 Framework 通过自己的调度器向 Mesos Master 注册，以接入到 Mesos 中；而 Mesos Slave 的主要功能是汇报任务的状态和启动各个 Framework 的 Executor（如 Hadoop 的 Executor 就是 TaskTracker）。

9.7.2 Apache Mesos 总体架构

Apache Mesos 的总体架构如图 9.21 所示。

^① 参考 <http://dongxicheng.org/>。

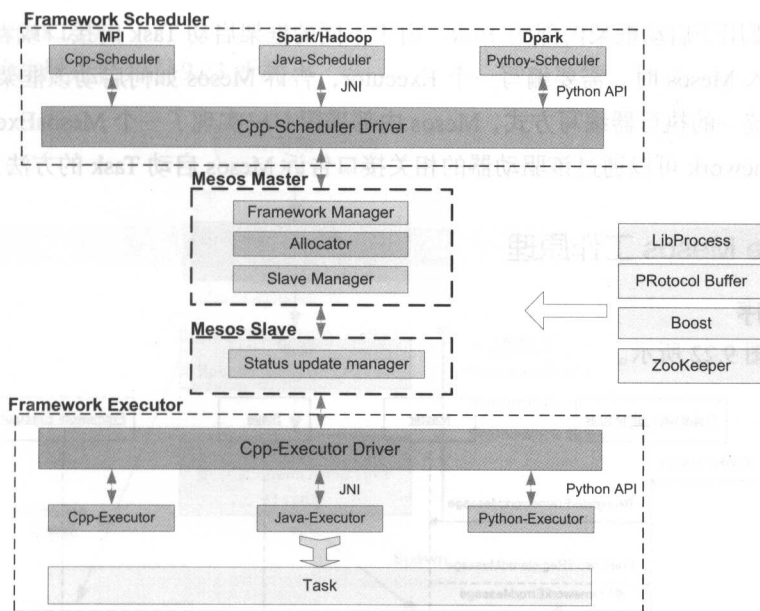


图 9.21

Apache Mesos 由 4 个组件组成，分别是 Mesos-master、Mesos-slave、Framework 和 Executor。

Mesos-master 是整个系统的核心，负责管理接入 Mesos 的各个 Framework（由 frameworks_manager 管理）和 Slave（由 slaves_manager 管理），并将 Slave 上的资源按照某种策略分配给 Framework（由独立插拔模块 Allocator 管理）。

Mesos-slave 负责接收并执行来自 Mesos-master 的命令，管理节点上的 Mesos-task，并为各个 Task 分配资源。Mesos-slave 将自己的资源发送给 Mesos-master，由 Mesos-master 中的 Allocator 模块决定将资源分配给哪个 Framework。当前考虑的资源有 CPU 和内存两种，也就是说，Mesos-slave 会将 CPU 个数和内存量发送给 Mesos-master；而用户在提交作业时，需要指定每个任务需要的 CPU 个数和内存量。这样，当任务运行时，Mesos-slave 会将任务放到包含固定资源的 Linux Container 中运行，以达到资源隔离的效果。很明显，Master 存在单点故障问题，为此，Mesos 采用 ZooKeeper 解决该问题。

Framework 是指外部的计算框架，如 Hadoop、Mesos 等，这些计算框架可以通过注册的方式接入 Mesos，以便 Mesos 进行统一管理和资源分配。Mesos 要求可接入的框架必须有一个调度器模块，该调度器模块负责框架内部的任务调度。当一个 Framework 想要接入 Mesos 时，需要修改自己的调度器，以便向 Mesos 注册，并获取 Mesos 分配给自己的资源，这样再由自己的调度器将这些资源分配给框架中的任务。也就是说，整个 Mesos 系统采用了双层调度框架：第一层，由 Mesos 将资源分配给框架；第二层，框架自己的调度器将资源分配给自己内部的任务。当前 Mesos 支持三种语言编写的调度器，分别是 C++、Java 和 Python。为了向各种调度器提供统一的接入方式，Mesos 内部采用 C++实现了一个 MesosSchedulerDriver（调度器驱动器），Framework 的调度器可调用该 Driver 中的接口与 Mesos-master 交互，完成一系列功能（如注册、资源分配等）。

Executor 主要用于启动框架内部的 Task。由于不同的框架启动 Task 的接口或者方式不同，当一个新的框架要接入 Mesos 时，需要编写一个 Executor，告诉 Mesos 如何启动该框架中的 Task。为了向各种框架提供统一的执行器编写方式，Mesos 内部采用 C++实现了一个 MesosExecutorDriver（执行器驱动器），Framework 可以通过该驱动器的相关接口告诉 Mesos 启动 Task 的方法。

9.7.3 Apache Mesos 工作原理

1. 整体时序

整体时序如图 9.22 所示。

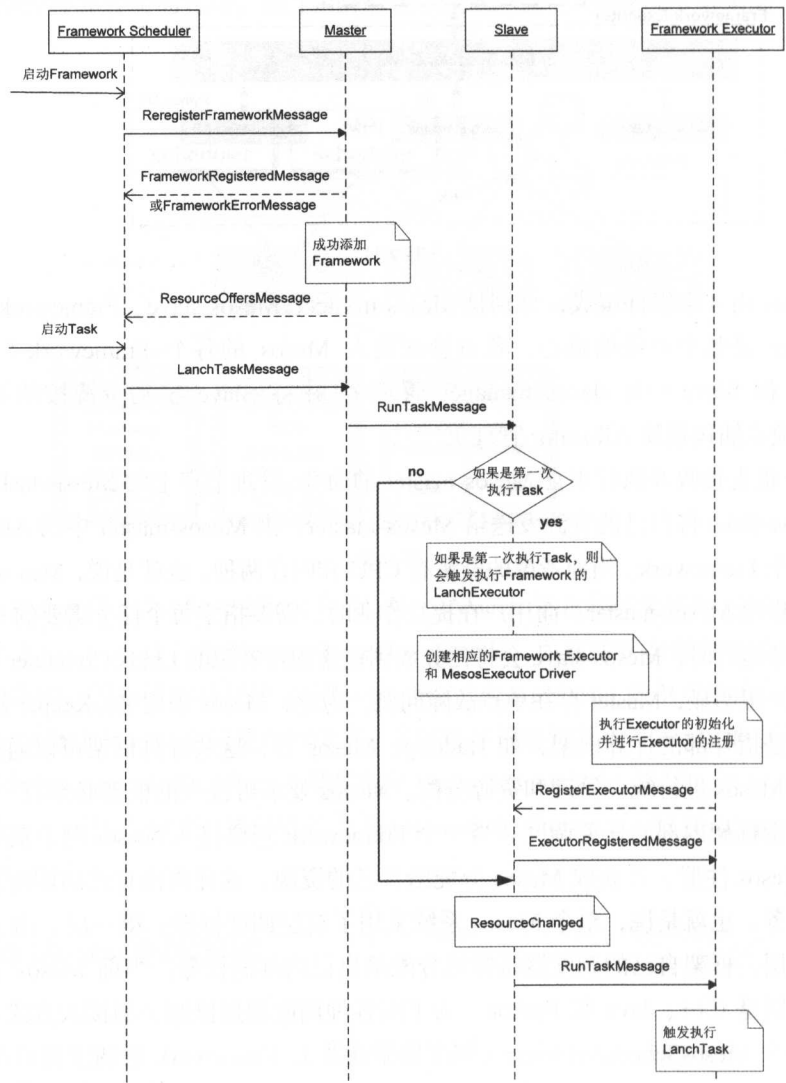


图 9.22

2. Framework 注册

Framework 的注册过程如图 9.23 所示。

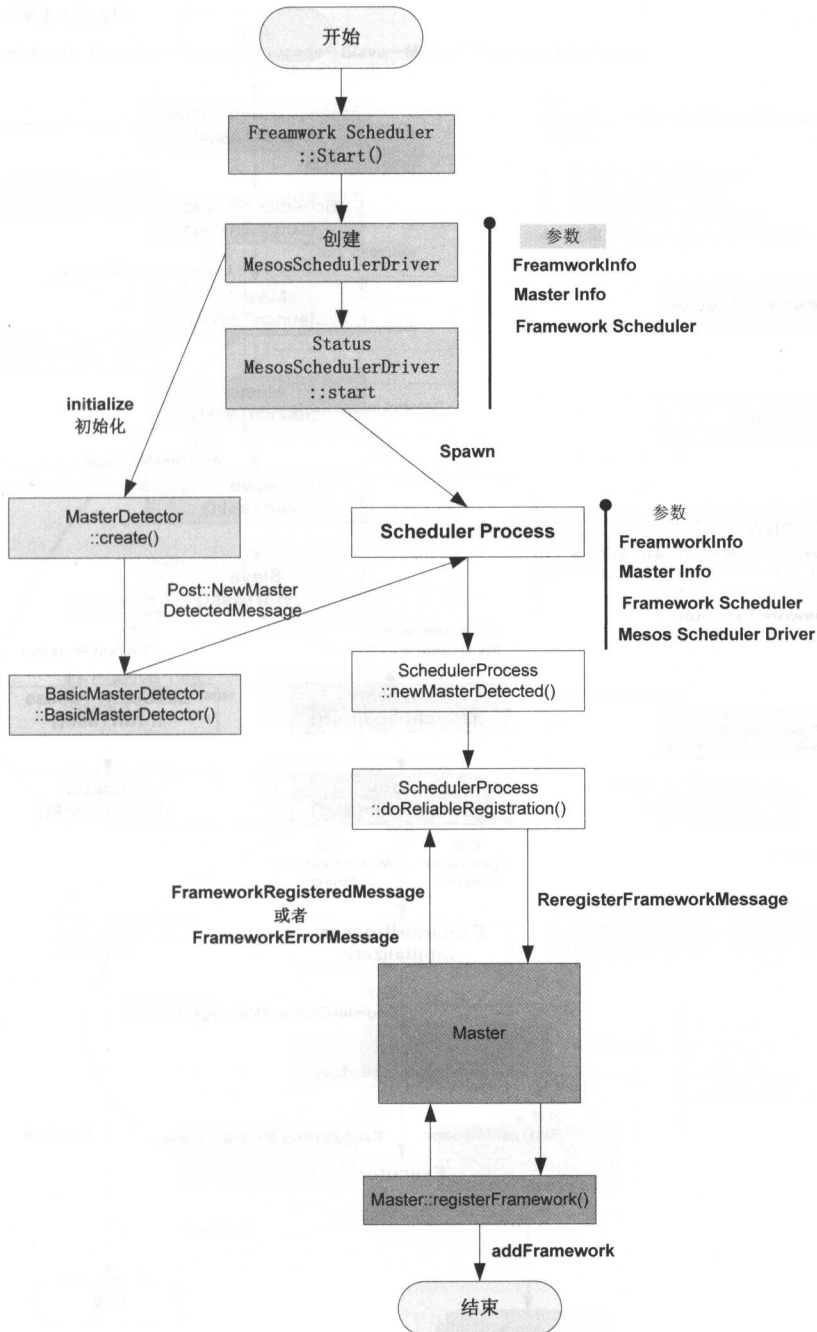


图 9.23

3. Executor 的注册

Executor 的注册过程如图 9.24 所示。

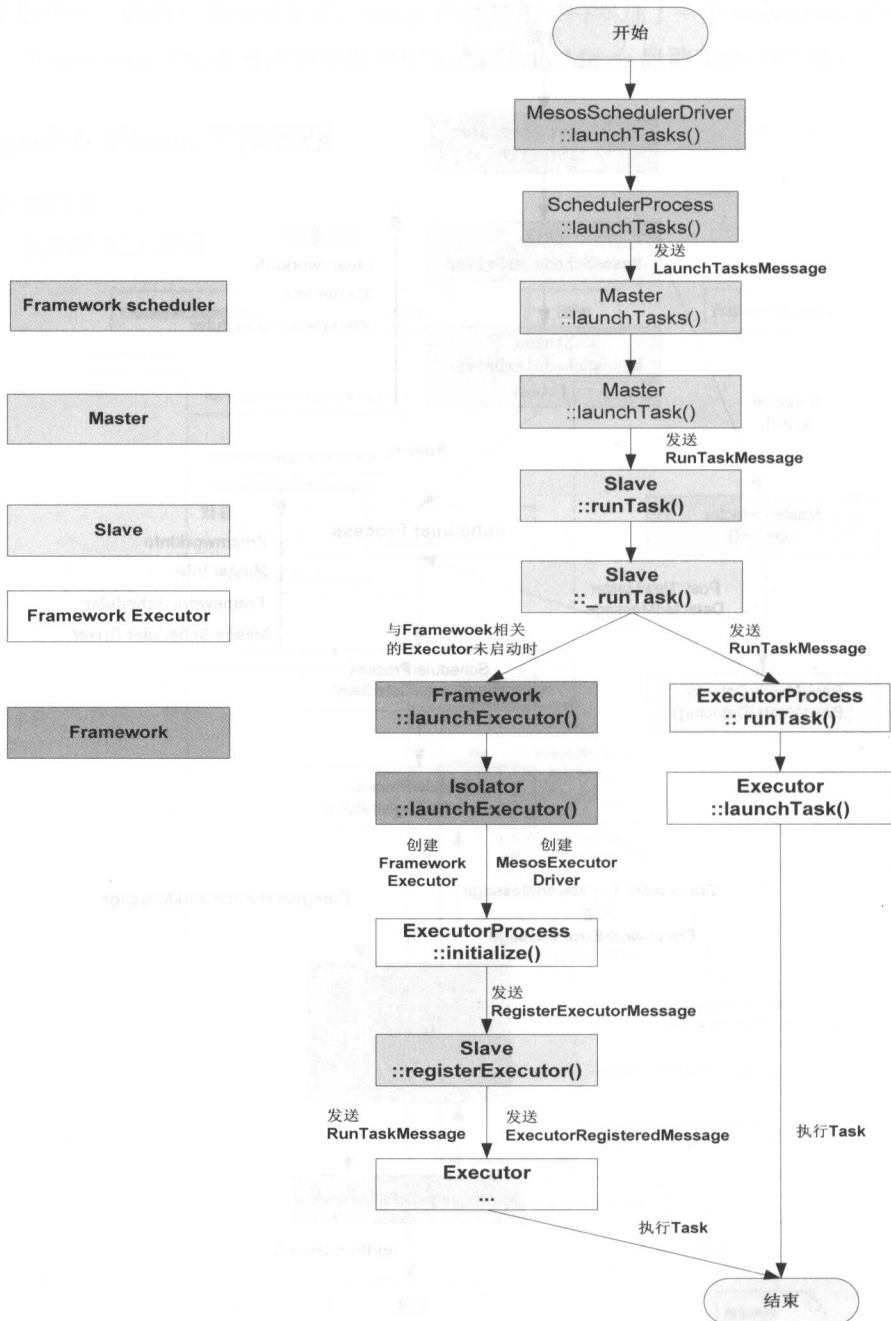


图 9.24

4. 状态更新

状态更新过程如图 9.25 所示。

任务状态上报流程：

Framework Executor → ExecutorProcess → Slave → Master → FrameworkScheduler

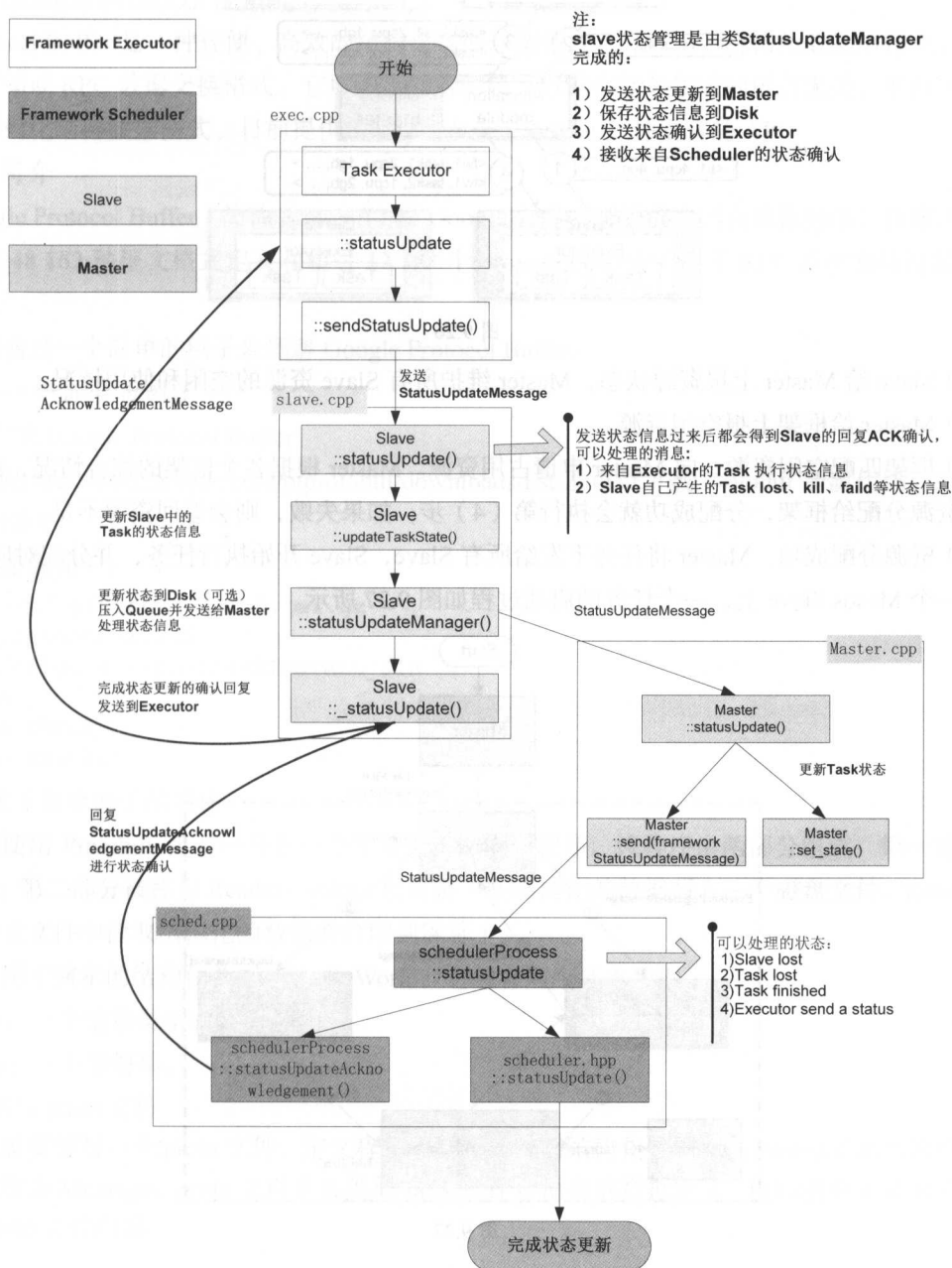


图 9.25

5. 任务分配

任务分配过程如图 9.26 所示。

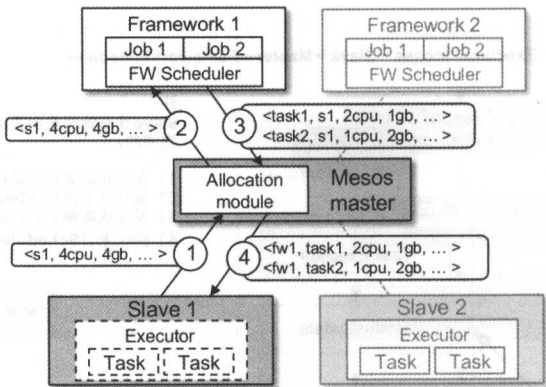


图 9.26

(1) Slave 给 Master 上报资源状态，Master 维护所有 Slave 资源的空闲和使用情况。

(2) Master 给框架上报空闲资源。

(3) 框架匹配空闲资源，向 Master 申请占用资源。Master 根据各个框架的综合情况，使用 DRF 算法把资源分配给框架，分配成功就会执行第 (4) 步；如果失败，则会返回资源不足。

(4) 资源分配成功，Master 将任务下发给所有 Slave，Slave 开始执行任务，并分配对应的资源。在一个 Mesos Slave 上，一个任务的启动过程如图 9.27 所示。

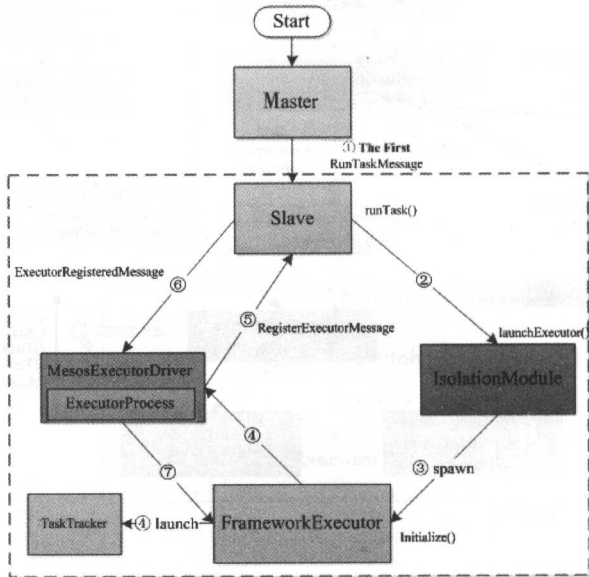


图 9.27

9.7.4 Apache Mesos 关键技术

Mesos 是一个资源管理框架，抽象来看就是一个框架+资源分配算法。前面已经介绍了资源分配算法，接下来重点看一下框架中用到的关键技术，分别是 Google Protocol Buffer 和 LibProcess。

1. Google Protocol Buffer^①

Protocol Buffer 是一种轻便、高效的结构化数据存储格式，可以用于结构化数据串行化，很适合做数据存储或 RPC 数据交换格式。它可用于通信协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。目前提供了 C++、Java、Python 三种语言的 API。

1) 简介

Google Protocol Buffer (简称 Protobuf) 是 Google 公司内部的混合语言数据标准，目前正在使用的有超过 48 162 种报文格式定义和超过 12 183 个 .proto 文件。它们用于 RPC 系统和持续数据存储系统。

下面通过一个简单的例子来理解 Google Protocol Buffer。

2) 一个简单的例子

① 安装 Google Protocol Buffer

在网站 <http://code.google.com/p/protobuf/downloads/list> 上可以下载 Protobuf 的源代码，然后解压编译安装即可。

安装步骤如下：

```
tar -xzf protobuf-2.1.0.tar.gz
cd protobuf-2.1.0
./configure --prefix=$INSTALL_DIR
make
make check
make install
```

② 关于简单例子的描述

下面使用 Protobuf 和 C++ 开发一个十分简单的例子程序。该程序由两部分组成：第一部分被称为 Writer；第二部分被称为 Reader。Writer 负责将一些结构化的数据写入一个磁盘文件，Reader 则负责从该磁盘文件中读取结构化的数据并打印到屏幕上。

准备用于演示的结构化数据是 HelloWorld，它包含两个基本数据。

- ID: 一个整数类型的数据。
- Str: 一个字符串。

③ 书写 .proto 文件

首先需要编写一个 .proto 文件，定义程序中需要处理的结构化数据。在 Protobuf 的术语中，结构化数据被称为 Message。 .proto 文件非常类似 Java 或者 C 语言的数据定义。代码清单 1 显示了例子应用中的 .proto 文件内容。

^① Google Protocol Buffer 的使用和原理，<http://www.ibm.com/developerworks/cn/linux/l-cn-gpb/index.html>。

```
package lm;
message helloworld
{
    required int32    id = 1;    // ID
    required string   str = 2;    // str
    optional int32    opt = 3;    //optional field
}
```

一个比较好的习惯是认真对待.proto 文件的文件名。比如将命名规则定义如下：

```
package Name.Message Name.proto
```

在上例中，package 的名字为 lm，定义了一个消息 helloworld，该消息有三个成员：类型为 int32 的 id；类型为 string 的成员 str；可选的成员 opt，即消息中可以不包含该成员。

④ 编译.proto 文件

编写好.proto 文件之后，就可以使用 Protobuf 编译器将该文件编译成目标语言了。本例中将使用 C++ 语言。

假设.proto 文件存放在 \$SRC_DIR 目录下，若想把生成的文件放在同一个目录下，则可以使用如下命令：

```
protoc -I=$SRC_DIR --cpp_out=$DST_DIR $SRC_DIR/addressbook.proto
```

该命令将生成两个文件：lm.helloworld.pb.h，定义了 C++ 类的头文件；lm.helloworld.pb.cc，定义了 C++ 类的实现文件。

在生成的头文件中，定义了一个 C++ 类 helloworld，后面的 Writer 和 Reader 将使用这个类来对消息进行操作，对消息成员进行赋值、将消息序列化等。

⑤ 编写 Writer 和 Reader

如前所述，Writer 把一个结构化数据写入磁盘，以便他人来读取。如果不使用 Protobuf，其实也有很多选择。一个可能的方法是将数据转换为字符串，然后将字符串写入磁盘。转换为字符串可以使用 sprintf() 方法。

这样做似乎没有什么不妥，但是仔细考虑一下就会发现，这样的做法对写 Reader 的人的要求比较高，Reader 的作者必须了解 Writer 的细节。比如“123”可以是单个数字 123，也可以是三个数字 1、2 和 3。这么说来，还必须让 Writer 定义一种类似分隔符的字符，以便 Reader 可以正确读取。但分隔符可能会引起其他问题。最后发现，一个简单的 helloworld 类也需要写很多处理消息格式的代码。

如果使用 Protobuf，那么这些细节就不需要考虑了。

使用 Protobuf，Writer 的工作很简单，需要处理的结构化数据由.proto 文件描述，经过编译之后，该数据化结构对应一个 C++ 的类，并定义在 lm.helloworld.pb.h 中。对于本例，类名为 lm::helloworld。

Writer 需要包含该头文件，以便使用这个类。

现在，在 Writer 代码中，将要存入磁盘的结构化数据由一个 lm::helloworld 类的对象表示，它提供了一系列的 get/set 函数来修改和读取结构化数据中的数据成员。

当需要将该结构化数据保存到磁盘上时，类 `lm::helloworld` 已经提供了相应的方法来把一个复杂的数据变成一个字节序列，可以将这个字节序列写入磁盘。

对于想要读取这个数据的程序来说，只需使用类 `lm::helloworld` 的相应反序列化方法来将这个字节序列重新转换为结构化数据即可。

Writer 的主要代码如下：

```
#include "lm.helloworld.pb.h"
...

int main(void)
{

    lm::helloworld msg1;
    msg1.set_id(101);
    msg1.set_str("hello");

    // Write the new address book back to disk.
    fstream output("./log", ios::out | ios::trunc | ios::binary);

    if (!msg1.SerializeToOstream(&output)) {
        cerr << "Failed to write msg." << endl;
        return -1;
    }
    return 0;
}
```

`msg1` 是一个 `helloworld` 类的对象，`set_id()` 用来设置 `id` 的值，`SerializeToOstream` 将对象序列化后写入一个 `fstream` 流。

Reader 的主要代码如下：

```
#include "lm.helloworld.pb.h"
...

void ListMsg(const lm::helloworld & msg) {
    cout << msg.id() << endl;
    cout << msg.str() << endl;
}

int main(int argc, char* argv[]) {

    lm::helloworld msg1;

    {
        fstream input("./log", ios::in | ios::binary);
        if (!msg1.ParseFromIstream(&input)) {
            cerr << "Failed to parse address book." << endl;
            return -1;
        }
    }
}
```

```
ListMsg(msg1);
```

```
...
```

```
}
```

同样，Reader 声明类 helloworld 的对象 msg1，然后利用 ParseFromIstream 从一个 fstream 流中读取信息并反序列化。此后，ListMsg 中采用 get 方法读取消息的内部信息，并执行打印输出操作。

⑥ 运行结果

运行 Writer 和 Reader 的结果如下：

```
>writer
>reader
101
Hello
```

Reader 读取文件日志中的序列化信息并打印到屏幕上。

这个例子本身并无意义，但只要稍加修改就可以将它变成更加有用的程序。比如，将磁盘替换为网络 Socket，那么就可以实现基于网络的数据交换任务。而存储和交换正是 Protobuf 最有效的应用领域。

3) Protobuf 的优点

Protobuf 类似 XML，不过它更小、更快、更简单。用户可以定义自己的数据结构，然后使用代码生成器生成的代码来读/写这个数据结构，甚至可以在无须重新部署程序的情况下更新数据结构。只需使用 Protobuf 对数据结构进行一次描述，即可利用各种不同语言或从各种不同数据流中对结构化数据进行轻松读/写。

它有一个非常棒的特性，即“向后”兼容性好，人们不必破坏已部署的、依靠“老”数据格式的程序就可以对数据结构进行升级。这样就可以不必担心因为消息结构的改变而造成大规模的代码重构或者迁移。

Protobuf 的语义更清晰，无须类似 XML 解析器的内容（因为 Protobuf 编译器会将 .proto 文件编译成对应的数据访问类，以对 Protobuf 数据进行序列化、反序列化操作）。

使用 Protobuf 无须学习复杂的文档对象模型。Protobuf 的编程模式比较友好、简单易学，同时它拥有良好的文档和示例。对于喜欢简单事物的人而言，Protobuf 比其他技术更有吸引力。

4) Protobuf 的不足

Protobuf 与 XML 相比也有不足之处。它功能简单，却无法用来表示复杂的概念。

XML 已经成为多种行业标准的编写工具，而 Protobuf 只是 Google 公司内部使用的工具，在通用性上相差很多。

由于文本并不适合用来描述数据结构，所以 Protobuf 也不适合用来对基于文本的标记文档（如 HTML）建模。另外，由于 XML 具有某种程度上的自解释性，它可以被直接读取编辑。而 Protobuf 以二进制的方式存储，除非有 .proto 定义，否则无法直接读出 Protobuf 的任何内容。

5) 高级应用话题

① 更复杂的 Message

到这里为止，我们只给出了一个简单的没有任何用处的例子。在实际应用中，人们往往需要定义更加复杂的 Message。我们用“复杂”这个词，不仅仅是指从个数上说有更多的 Fields 或者更多类型的 Fields，而是指更加复杂的数据结构。

② 嵌套 Message

嵌套是一个神奇的概念，一旦拥有嵌套能力，消息的表达能力就会非常强大。如下代码给出一个嵌套 Message 的例子：

```
message Person {
  required string name = 1;
  required int32 id = 2;          // Unique ID number for this person.
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }
  repeated PhoneNumber phone = 4;
}
```

在 messagePerson 中定义了嵌套消息 PhoneNumber，并用来定义 Person 消息中的 phone 域，这使得人们可以定义更加复杂的数据结构。

③ ImportMessage

在一个 .proto 文件中，还可以用 Import 关键字引入在其他 .proto 文件中定义的消息，这可以称为 ImportMessage，或者 DependencyMessage。

比如：

```
import common.header;

message youMsg{
  required common.info_header header = 1;
  required string youPrivateData = 2;
}
```

其中，common.info_header 定义在 common.header 包内。

ImportMessage 的用处主要在于提供了方便的代码管理机制，类似 C 语言中的头文件。用户可以将一些公用的 Message 定义在一个 package 中，然后在其他 .proto 文件中引入该 package，进而使用其中的消息定义。

Google Protocol Buffer 可以很好地支持嵌套 Message 和引入 Message，从而让定义复杂的数据结构的工作变得轻松愉快。

④ 动态编译

一般情况下，Protobuf 的使用者都会先写好 .proto 文件，再用 Protobuf 编译器生成目标语言所需的源代码文件，然后将这些生成的代码和应用程序一起编译。

但在某些情况下，人们无法预先知道 .proto 文件，他们需要动态处理一些未知的 .proto 文件。比如一个通用的消息转发中间件，它不可能预知需要处理怎样的消息。这就需要动态编译 .proto 文件，并使用其中的 Message。

Protobuf 提供了 `google::protobuf::compiler` 包来完成动态编译的功能。主要的类叫作 `Importer`，定义在 `importer.h` 中。使用 `Importer` 非常简单，图 9.28 展示了 `Importer` 和其他几个重要类的关系。

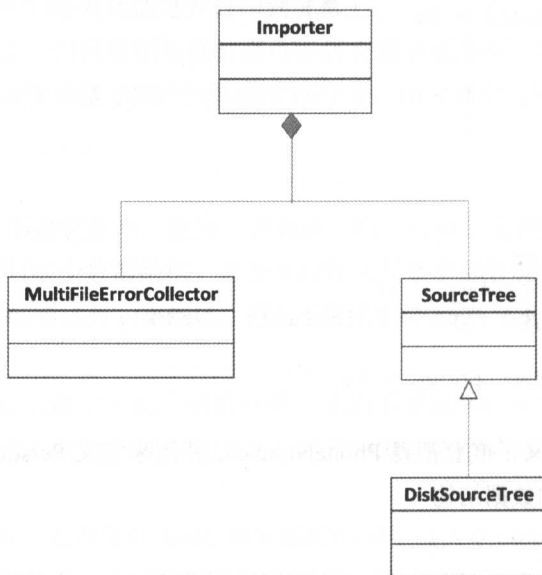


图 9.28

`Importer` 类中包含三个主要的对象，分别为处理错误的 `MultiFileErrorCollector` 类，定义 .proto 文件源目录的 `SourceTree` 类，以及 `SourceTree` 对应的派生类 `DiskSourceTree`。

下面通过实例说明这些类的关系和使用。

对于给定的 .proto 文件，如 `lm.helloworld.proto`，在程序中动态编译它只需很少的代码。示例代码如下：

```

google::protobuf::compiler::MultiFileErrorCollector errorCollector;
google::protobuf::compiler::DiskSourceTree sourceTree;

google::protobuf::compiler::Importer importer(&sourceTree, &errorCollector);
sourceTree.MapPath("", protosrc);

importer.import("lm.helloworld.proto");
  
```

首先构造一个 `importer` 对象。构造函数需要两个入口参数：一个是 `sourceTree` 对象，该对象指定了存放 `.proto` 文件的源目录；另一个参数是一个 `errorCollector` 对象，该对象有一个 `AddError()` 方法，用来处理解析 `.proto` 文件时遇到的语法错误。

之后，当需要动态编译一个 `.proto` 文件时，只需调用 `importer` 对象的 `import()` 方法即可。

那么，如何使用动态编译后的 `Message` 呢？首先需要了解几个其他的类。`Packagegoogle::protobuf::compiler` 中提供了以下几个类，用来表示一个 `.proto` 文件中定义的 `Message`，以及 `Message` 中的 `Field`。其中，类 `FileDescriptor` 表示一个编译后的 `.proto` 文件；类 `Descriptor` 对应该文件中的一个 `Message`；类 `FieldDescriptor` 描述一个 `Message` 中的一个具体的 `Field`。

编译完 `lm.helloworld.proto` 后，可以通过如下代码得到 `lm.helloworld.id` 的定义：

```
const protobuf::Descriptor *desc =
    importer_.pool()->FindMessageTypeByName("lm.helloworld");
const protobuf::FieldDescriptor* field =
    desc->pool()->FindFileByName ("id");
```

通过 `Descriptor`、`FieldDescriptor` 的各种方法和属性，应用程序可以获得各种关于 `Message` 定义的信息。比如，通过 `field->name()` 得到 `Field` 的名字。这样就可以使用一个动态定义的消息了。

⑤ 编写新的 `protoc` 编译器

随 Google Protocol Buffer 源代码一起发布的编译器 `protoc` 支持 3 种编程语言：C++、Java 和 Python。但使用 Google Protocol Buffer 的 `Compiler` 包，可以开发出支持其他语言的新的编译器。

类 `CommandLineInterface` 封装了 `protoc` 编译器的前端，包括命令行参数的解析、`.proto` 文件的编译等功能。用户需要做的是实现类 `CodeGenerator` 的派生类，实现诸如代码生成等后端工作。

在 `main()` 函数内生成 `CommandLineInterface` 的对象 `cli`，调用其 `RegisterGenerator()` 方法将新语言的后端代码生成器 `yourG` 对象注册给 `cli` 对象，然后调用 `cli` 的 `Run()` 方法即可。

这样生成的编译器和 `protoc` 编译器的使用方法相同，接受同样的命令行参数，`cli` 将对用户输入的 `.proto` 文件进行词法、语法等分析工作，最终生成一棵语法树。

其根节点为一个 `FileDescriptor` 对象，并作为输入参数被传入 `yourG` 的 `Generator()` 方法。在这个方法内，可以遍历语法树，然后生成对应的代码。简单说来，要想实现一个新的 `Compiler`，只需写一个 `main()` 函数和一个实现了 `Generator()` 方法的派生类即可。

6) Protobuf 的更多细节

人们一直在强调，同 XML 相比，Protobuf 的主要优点在于性能高。它以高效的二进制方式存储，比 XML 小 3~10 倍，快 20~100 倍。

有两项技术保证了采用 Protobuf 的程序能获得相对于 XML 极大的性能提高。

首先，可以考察 Protobuf 序列化后的信息内容。可以看到 Protobuf 信息的表示非常紧凑，这意味着消息的体积减小，自然需要更少的资源，如网络上传输的字节数更少、需要的 I/O 更少等，从而提高性能。

其次，需要理解 Protobuf 封装解包的大致过程，从而理解它为什么会比 XML 快很多。

7) Google Protocol Buffer 的 Encoding

Protobuf 序列化后生成的二进制消息非常紧凑，这得益于 Protobuf 采用了非常巧妙的 Encoding 方法。

在考察消息结构之前，首先介绍一个名为 Varint 的术语。

Varint 是一种紧凑地表示数字的方法。它用一个或多个字节来表示一个数字，值越小的数字使用越少的字节数，这样就能减少用来表示数字的字节数。

消息经过序列化后会成为一个二进制数据流，该流中的数据为一系列的 Key/Value 对。

采用这种 Key/Value 结构无须使用分隔符来分割不同的 Field。对于可选的 Field，如果消息中不存在该 Field，那么在最终的 MessageBuffer 中就没有该 Field。这些特性都有助于减少消息本身的大小。

假设我们生成如下的一个消息 Test1：

```
Test1.id=10;  
Test1.str="hello";
```

则最终的 MessageBuffer 中有两个 Key/Value 对，一个对应消息中的 id，另一个对应 str。Key 用来标识具体的 Field，在解包的时候，Protobuf 根据 Key 就可以知道相应的 Value 应该对应消息中的哪一个 Field。

Key 的定义如下：

```
(field_number<<3)|wire_type
```

可以看到，Key 由两部分组成：第一部分是 field_number，如消息 lm.helloworld 中 fieldid 的 field_number 为 1；第二部分为 wire_type，表示 Value 的传输类型。

8) 封解包的速度

首先来了解一下 XML 的封解包过程。XML 需要先从文件中读取字符串，再转换为 XML 文档对象结构模型，然后再从 XML 文档对象结构模型中读取指定节点的字符串，最后将这个字符串转换成指定类型的变量。这个过程非常复杂，其中将 XML 文件转换为文档对象结构模型的过程通常需要完成词法、文法分析等大量消耗 CPU 的复杂计算。

反观 Protobuf，只需简单地将一个二进制序列按照指定的格式读取到 C++ 对应的结构类型中即可，速度非常快。

2. LibProcess

1) 背景

LibProcess 是一套基于 Socket 实现的通信协议库，它支持 Protobuf，二者相结合，可实现一套很高效的基于消息传递的通信协议库，而 Mesos 底层通信协议正好采用了该库。

2) 一个简单的实例

假设有两个服务 Master 和 Slave，Slave 周期性地向 Master 汇报自己的进度，而 Master 则不定期地向 Slave 下达任务。采用 LibProcess 实现如下。

(1) Master 类设计。

步骤 1: 继承 ProtobufProcess 类。让 Master 类继承 LibProcess 中的 ProtobufProcess 类, 该类实际上维护了一个 SocketServer, 负责处理实现注册好的各种 ProtocolBuffer 定义的 Message。

```
class Master:public ProtobufProcess<Master>{
//.....
}
```

步骤 2: 注册消息处理器。在初始化函数 initialize() 中注册一个 ReportProgressMessage 类型的消息处理器, 该消息处理器对应的函数是 Master::reportProgress, 该函数带有一个参数 tasks。这样, Master 内部的 SocketServer 会监听来自外部的各种消息包, 一旦发现 ReportProgressMessage 类型的消息包, 就会调用函数 reportProgress 进行处理。注意, reportProgress 函数中的参数必须对应 ReportProgressMessage 消息的定义。

```
void initialize(){ //继承 ProtobufProcess 类后, 需实现该函数
install<ReportProgressMessage>( //使用 install 函数进行注册
&Master::reportProgress,
&LaunchTasksMessage::tasks);
}
```

步骤 3: 定义 ReportProgressMessage 消息。使用 Protobuf 定义 ReportProgressMessage 消息如下:

```
//master.proto
message ReportProgressMessage {
repeated Task tasks = 3;
}
message Task {
required string name = 1;
required TaskID task_id = 2;
required float progress = 3;
}
message TaskID {
required string value = 1;
}
```

步骤 4: 编写消息处理器 reportProgress。

```
void reportProgress(const vector<Task>& tasks) {
for (int i = 0; i < tasks.size(); i++) {
//update tasks[i] information;
}
```

步骤 5: 编写 main 函数启动 Master。

```
int main(int argc, char** argv)
{
process::initialize("master"); //初始化一个名为 master 的进程
Master* master = new Master();
process::spawn(master); //启动 master, 实际上是一个 SocketServer
process::wait(master->self());
delete master;
return 0;
}
```

比较全的代码如下：

```
class Master : public ProtobufProcess<Master>           //步骤 1
{
Master(): ProcessBase("master") {}
void initialize() {
install<ReportProgressMessage>(                        //步骤 2
&Master::reportProgress,
&LaunchTasksMessage::tasks);
}
void reportProgress(const vector<Task>& tasks) {        //步骤 4
for (int i = 0; i < tasks.size(); i++) {
//update tasks[i] information;
}
}
}
```

(2) Slave 类设计。

Slave 类设计与 Master 类设计类似，具体过程如下：

```
class Slave : public ProtobufProcess<Slave>
{
Slave(): ProcessBase("slave") {}
void initialize() [
install<LaunchTasksMessage>(
&Master::launchTasks,
&LaunchTasksMessage::id,
&LaunchTasksMessage::tasks);
}
void launchTasks(const int id,
const vector<TaskInfo>& tasks) {
for (int i = 0; i < tasks.size(); i++) {
//launch tasks[i];
}
}
}
```

3) LibProcess 事件驱动模型

LibProcess 采用了基于事件驱动的编程模型，每个服务（进程）内部实际上运行了一个 SocketServer，而不同服务之间通过消息（事件）进行通信。在一个服务内部注册了很多消息及每个消息对应的处理器，一旦收到某种类型的消息，则会调用相应的处理器进行处理，在处理过程中可能会产生另外一种消息发送给另一个服务。

9.7.5 Mesos 与 YARN 比较

Mesos 与 YARN 主要在以下几个方面有明显不同：

(1) 框架担任的角色。在 Mesos 中，各种计算框架是完全融入 Mesos 中的，也就是说，如果想在 Mesos 中添加一个新的计算框架，首先需要在 Mesos 中部署一套该框架；而在 YARN 中，各种框

架作为 Client 端的 Library 使用, 仅仅是用户编写的程序的一个库, 不需要事先部署一套该框架。从这方面来说, YARN 运行和使用起来更加方便。

(2) 调度机制。Mesos 采用双层调度策略, 第一层是 Mesos Master 将空闲资源分配给某个框架, 第二层是计算框架自带的调度器对分配到的空闲资源进行分配, 也就是说, Mesos 将大部分调度任务授权给了计算框架; 而 YARN 是一个单层调度架构, 各种框架的任务一视同仁, 全部由 Resource Manager 进行统一调度。总的来说, MesosMaster 首先完成粗粒度的资源分配, 即将资源分配给框架, 然后由框架进行细粒度的资源分配; 而 ResourceManager 直接进行细粒度的分配, 即直接将资源分配给某个任务 (Task)。

其他特性对比如表 9.2 所示。

表 9.2

	Mesos	YARN (Hadoop-0.23.0)
开源协议	Apache License 2.0	Apache License 2.0
支持的框架	在线计算、离线计算、HPC 等	在线计算、离线计算、HPC 等
隔离性	采用 Linux Container 对内存和 CPU 两种资源进行隔离	当前实现仅采用 OS 进程级别隔离 (方法与 Hadoop 类似)
容错性	ZooKeeper 对 Master 进行容错; Slave 失败任务转移; 各种框架自身的容错机制	ZooKeeper 对 ResourceManager 进行容错; ApplicationMaster 监控任务执行, 定期将任务执行信息保存 to 磁盘上, 如果失败重新则为任务申请资源; ResourceManager 负责监控, 让 Application Master 失败后重启并恢复各个任务运行状态

9.8 小结

本章主要介绍了大数据领域资源管理的一些基础知识, 以及 Mesos 面向数据中心的 DCOS 思路。从目前来看, 资源管理和调度还有大量不成熟的问题, 在一定时期内会有多套框架和解决方案共存, 如 Mesos 和 YARN, 相互之间有一定的重叠和差异化。

从未来的趋势来看, 资源管理和调度应该分为两个层面: 一个是颗粒度的、面向数据中心层面的整体资源调度和管理, 会有一个统一的框架; 另一个层面是任务级别的调度, 称为小颗粒调度, 未来会聚焦于应用本身的特征, 基于应用特征进行智能的资源管理和分配, 以获取高效的资源利用。

从数学角度来说, 资源管理和分配也是一个很难解决的问题, 未来还需要相当长时间的完善和发展。

第 10 章

存储是基础

存储是所有大数据组件的基础，存储的发展远低于 CPU 和 MEM，导致 CPU 和存储的速度差越来越大，所以对于 DBA 来说，调优有时候基本等价于调存储。

本章将从系统架构和应用角度讲述对存储的理解，希望对读者理解存储有所帮助。

10.1 分久必合，合久必分

“分久必合，合久必分”，这句话形容存储和服务器的关系最为恰当。在计算机发明之初，存储是外置的，如打孔纸带；后来发明了硬盘，硬盘是装到计算机内部的；硬盘用了一段时间，发现容量不够，于是就有了 SAN 存储，放到计算机外面；云计算时代的到来，又产生了 HDFS，通过软件聚合普通服务器内部的硬盘，对外提供统一的存储，以及诞生了所谓的 Server SAN，即软件定义存储。

下面从存储的硬件形态、存储的接口及衡量存储的两个关键指标（吞吐量、IOPS）来讲解为解决存储效率问题而发展起来的索引技术。

10.2 存储硬件的发展

本节先介绍机械硬盘的工作原理，之后讲述逐渐成熟的 SSD 技术。可见的将来，SSD 会逐渐替代硬盘成为主流。

10.2.1 机械硬盘的工作原理

图 10.1 是一款双碟的机械硬盘。任何机械硬盘的结构都是一样的：电路板上的主控制器芯片负责与芯片组之间的通信，并且控制硬盘内部的运转；盘片是用磁性材料制成的，固定在硬盘中部的电动机上旋转（这里就有了转速的区别：5400rpm 指的是每分钟盘片旋转 5400 转，7200rpm 则表示每分钟旋转 7200 转）；磁头（图中近似于三角形的部件）则沿着盘片的径向移动。磁头的移动过程就是硬盘寻道的过程（这句话不太严谨，但是除了断电归位等情况外，绝大部分情况下都是如此）。至于“寻道”，则和盘片的结构有关。

盘片上划分为一圈一圈的同心圆环，每个圆环代表一个磁道，如图 10.2 所示。早期的机械硬盘

从圆心出发向四周发散出角间距相等的一系列直线（实际上并没有直线存在），直线与同心圆围成的最小区域就是一个扇区。这样的划分，在硬盘容量不大的年代还是简单易行的，但是随着硬盘技术的进步，磁道的划分越来越密集，必然导致外圈的扇区物理长度远远大于内圈的扇区，从而造成浪费。所以现在的硬盘都不用圆心发散的直线来划分扇区了，而是从外圈磁道开始取一定长度作为一个扇区，然后从外向里一个一个编号下去。这个编号就是扇区的地址，我们要确定文件在哪里全靠这个地址。扇区都有固定的大小，一般是 512 字节，现在的支持先进格式化的硬盘都采用 4096 字节作为一个扇区。

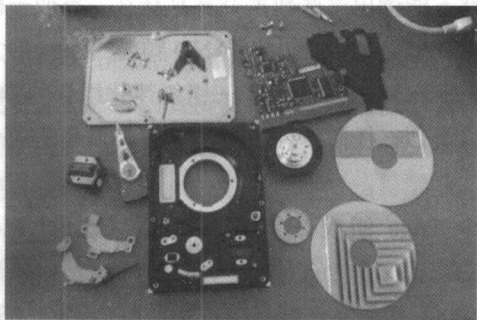


图 10.1

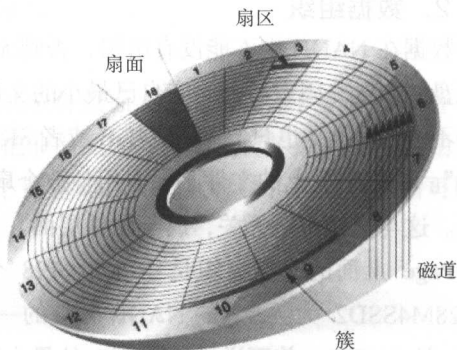


图 10.2

机械硬盘要读出数据，必须由磁头找到对应的磁道和扇区（对于多碟的机械硬盘来说，首先需要确定柱面），这全部依靠磁头的驱动电动机来驱动（磁头本身是依靠盘片旋转产生的气流来悬浮的）。电动机等机械装置的反应速度毕竟不快，所以机械硬盘会浪费大量的时间用于寻道操作（每次寻道大约需要 10ms）。尤其是对于零碎的小文件读/写来说，由于文件所在扇区不连续，需要不断地进行寻道，从而导致硬盘性能急剧下降。但是对于持续读/写来说，由于不需要进行不断地寻道，所以不存在寻道时间。所以机械硬盘的随机读/写能力很差（不超过 0.1MB/s），但是持续读/写能力并不差，而且随着单碟容量的提升和磁盘阵列的组建，持续读/写速度比固态硬盘更快。

10.2.2 SSD 的原理

固态硬盘（Solid State Drive, SSD）是由固态电子存储芯片阵列而制成的硬盘，由控制单元和存储单元（Flash 芯片、DRAM 芯片）组成。

1. NAND 颗粒

NAND 颗粒之间的关系类似于 RAID 0。NAND 是半导体存储颗粒的一种，至于这个颗粒的物理结构如何，没有必要深究，我们关心的是 NAND 如何存储和读取数据。

首先需要知道 NAND 是怎么制造的。制造 NAND 和制造 CPU 等处理器类似，都是使用高纯度硅，切割成晶圆之后使用光刻机和化学溶剂将设计好的电路蚀刻上去，然后用金属材料“镶嵌”而成的。这样制作出来的是一个布满 NAND 芯片的晶圆。将晶圆切开，然后对切割后的芯片精心挑选

测试，封装后就可以出厂了。整个工艺虽然和制造 CPU 类似，但是在电路等方面简单不少。

简单地说，NAND 可以视为由很多个电容器组成的集成电路。NAND 可以分为 SLC (Single-Level Cell)、MLC (Multi-Level Cell)、TLC (Trinary-Level Cell) 等。

其中，TLC 应用最少；MLC 技术是今后 NAND Flash 的发展趋势，就像 CPU 单核心、双核心、四核心一样，MLC 技术通过每 Cell (单元) 存储更多的 bit 来实现容量上的成倍跨越，直至更先进的架构问世；而 SLC 短期内仍然是市场的佼佼者，但随着 MLC 技术的不断发展和完善，SLC 必将退出历史舞台。

2. 数据组织

数据在 NAND 中不能没有组织，否则无论什么设备都不能读出这些数据究竟是什么。类似于机械硬盘的扇区，固态硬盘也有自己最小的文件存储单位，叫作 Page。目前的 NAND 颗粒，Page 的大小并不一致，但是也只有两种：4KB 或者 8KB (1KB=1024 字节)。Page 相当于一组规律化组合的“电容器”。机械硬盘上的文件必须占用整数个扇区；固态硬盘类似，任何文件占用的空间必须是整数个 Page。这与文件系统无关，属于硬件层面。

Page 上面的一层组织是 Block。128 个或者 256 个 Page 组成一个 Block。以 Crucial M4-CT128M4SSD2^①为例，它的 NAND 颗粒的一个 Page 是 4KB，128 个 Page 组成一个 Block，所以一个 Block 是 512KB。前面说过，NAND 的最小写入单位是 Page，任何文件都必须占用整数个 Page。这里的 Block 也有类似的作用：任何擦除（“电容器”放电）都必须是整数个 Block。也就是说，要擦除 NAND 里面存储的信息，每次最少擦除一个 Block，也可以擦除任意整数（当然不能是负数）个 Block。一定数量（2 的幂次方）的 Block 构成更高级的结构 Plane，两个 Plane 组成一个 Die，这就是我们看到的一个芯片。

NAND 有一个特性：如果要读取 NAND 中的信息，那么速度会很快；但是，如果要给 NAND 写入信息，尤其是 NAND 原有的信息需要覆盖的时候，速度会非常慢。但无论是读还是写，操作系统本身甚至计算机本身都是无法控制 NAND 芯片的。操作 NAND 芯片需要借助主控制器芯片，它也是影响 SSD 性能的重要因素之一。

3. 主控制器芯片

任何存储设备都有主控制器芯片，否则主板南桥芯片（或者 Intel 现在使用的单芯片组）无法直接与存储层进行通信。

主控制器芯片大多数是一个 ARM 处理器。SandForce 和 Marvell 的主控制器芯片一般是双核心 ARM 处理器，Samsung830 系列 SSD 上的主控制器芯片则是三核心的。

主控制器芯片负责与芯片组之间的通信，接受 ATA 指令与数据，并负责将数据写入 NAND 或者从 NAND 读出。

这里会发现 SSD 完全是类似 CPU 的芯片，不再受限于机械结构，所以 SSD 硬盘的容量和速度

^① 美光的一款 SSD。

也类似 CPU 的摩尔定律，发展非常快。相信不久 SSD 就会成为主流，硬盘会退化成今天磁带的角色。

10.2.3 3DXPoint

Intel 与美光发明的 3DXPoint 完全颠覆了 1989 年推出的 NAND 闪存存储器。下面对其进行简单介绍。

1. 3DXPoint 的工作原理

3DXPoint 抛弃了 NAND 芯片的核心——晶体管。NAND 的工作原理是运动的电子在称为“浮动栅”的晶体管内来回跑动，以此来表示二进制代码的 0 与 1。

2. 3DXPoint 的应用场景

3DXPoint 有三种应用模式：一是作为内存的扩展，从软件层面来看非常简单，不需要额外开发或者优化，可以直接使用；二是类似于 NVMe 的拓展，把非易失闪存存储进行扩展；三是应用层面的扩展，用户需要进行软件优化，定义哪些可以做内存、哪些可以做永久存储，在掉电之后可以实现自我保护。

10.2.4 硬件发展小结

从硬盘到 SSD 再到最新的 3DXPoint，硬件的发展速度越来越快，传统的性能瓶颈及调优方法随着硬件的发展不复存在。同时对软件架构来说也是一个挑战，因为软件要跟随硬件的发展做出相应的变化。

10.3 存储关键指标

衡量硬件的性能一般有如下 4 个指标。

- 容量：决定因子是硬盘个数和单盘容量。
- 吞吐量：决定因子是阵列架构、光纤通道大小和硬盘个数。
- IOPS：决定因子是磁盘个数、Cache 命中率和阵列算法。
- I/O 响应时间： $R=T/(1-U)$ 。其中， R 是响应时间， T 是 I/O 控制器服务一个块所用的时间， U 是硬盘利用率。

10.4 RAID 技术

磁盘阵列（Redundant Arrays of Independent Disks, RAID）是指由独立磁盘构成的具有冗余能力的阵列。磁盘阵列是由很多价格较便宜的磁盘组合成一个容量巨大的磁盘组，利用各磁盘提供的数据所产生的加成效果来提升整个磁盘系统效能。利用这项技术，可将数据切割成许多区段，分别存放在各个硬盘上。

磁盘阵列具有同位检查（Parity Check）能力，即在磁盘组中的任意一块硬盘出现故障时，仍可读出数据；在数据重构时，将数据经计算后重新置入新硬盘中。

磁盘阵列有三种样式：一是外接式磁盘阵列柜；二是内接式磁盘阵列卡；三是利用软件进行仿真。

磁盘阵列作为独立系统在主机外直连或通过网络与主机相连。磁盘阵列有多个端口，可以被不同主机或不同端口连接。一个主机连接阵列的不同端口可以提升传输速度。

在应用中，有部分常用的数据是需要经常读取的，磁盘阵列根据内部的算法查找出这些经常读取的数据，并存储在缓存中，加快主机读取这些数据的速度；而对于其他缓存中没有的数据，主机要想读取，则由阵列从磁盘上直接读取后传输给主机。对于主机写入的数据，只写在缓存中，主机可以立即完成写操作，然后由缓存再慢慢写入磁盘。

RAID 通过在多个磁盘上同时存储和读取数据来大幅提高存储系统的数据吞吐量（Throughput）。在 RAID 中，可以让很多磁盘驱动器同时传输数据，而这些磁盘驱动器在逻辑上又是一个磁盘驱动器，所以使用 RAID 可以达到单个磁盘驱动器几倍、几十倍甚至上百倍的速率。这也是 RAID 最初想要解决的问题。

RAID 通过数据校验提供容错功能。普通磁盘驱动器无法提供容错功能。RAID 容错是建立在每个磁盘驱动器的硬件容错功能之上的，所以它提供了更高的安全性。在很多 RAID 模式中都有较为完备的相互校验/恢复的措施，从而大大提高了 RAID 系统的容错度，提高了系统的稳定冗余性。

RAID 分为很多级别：RAID 0 没有冗余功能，如果一块磁盘（物理）损坏，则所有的数据都无法使用；RAID 1 磁盘的利用率最高只能达到 50%，是所有 RAID 级别中最低的；RAID 0+1 可以理解为 RAID 0 和 RAID 1 的折中方案，它可以为系统提供数据安全保障，但保障程度要比 Mirror 低，而磁盘空间利用率要比 Mirror 高。

提高 HDD（硬盘驱动器）性能的方法之一是组建磁盘阵列。通常用于提高性能的磁盘阵列是 RAID 0。例如，使用两块硬盘组建 RAID 0 之后，当有数据从芯片组传输给硬盘时，这个数据会被自动划分为两部分，每块硬盘各自存储一部分，这样一来，在理想状态下，RAID 0 的写入速度是翻倍的；读取也是类似的原理，每块硬盘拿出各自的数据，在理想状态下，读取速度也是翻倍的。但是 RAID 0 对于随机读/写并没有什么明显作用，所以要凭借组建 RAID 0 来提高系统盘的性能很困难。

10.5 存储接口

存储的分类方法有很多种，根据存储所处的位置可分为内置存储和外挂存储。开放系统的外挂存储根据连接的方式可分为直连式存储（Direct-Attached Storage, DAS）和网络化存储（Fabric-Attached Storage, FAS）；开放系统的网络化存储根据传输协议又可分为网络接入存储（Network-Attached Storage, NAS）和存储区域网络（Storage Area Network, SAN）。

从接口来看存储，又分为文件、对象、块、K/V 等典型接口，分别应对不同的应用。

很多人误以为存储主要靠硬件，其实存储是软硬件结合的技术，软件在存储里面占了很大的比重和作用。下面要讲的各种存储技术都是以软件为主、以硬件为辅实现的。

10.5.1 文件接口

说到文件接口，首先想到的是文件系统。

文件系统是操作系统用于明确磁盘或分区上的文件的方法和数据结构，即在磁盘上组织文件的方法。操作系统中负责管理和存储文件信息的软件机构称为文件管理系统，简称文件系统。文件系统由三部分组成：文件系统的接口、对对象操纵和管理的软件集合、对象及属性。

少数程序直接对磁盘或分区的原始扇区进行操作，这可能会破坏一个存在的文件系统。一个磁盘或分区在作为文件系统使用前需要初始化，并将记录数据结构写到磁盘上。这个过程叫作建立文件系统。

文件系统的功能包括：管理和调度文件的存储空间；提供文件的逻辑结构、物理结构和存储方法；实现文件从标识到实际地址的映射；实现文件的控制操作和存取操作；实现文件信息的共享并提供可靠的文件保密和保护措施；提供文件的安全措施。文件的逻辑结构是依照文件内容的逻辑关系来组织文件结构的，可以分为流式文件和记录式文件。

- 流式文件：文件中的数据是一串字符流，没有结构。
- 记录式文件：由若干逻辑记录组成，每条记录又由相同的数据项组成，数据项的长度可以是确定的，也可以是不确定的。

主要缺点：数据关联差，数据不一致，冗余度高。

10.5.2 裸设备

裸设备也叫裸分区（原始分区），是一种没有经过格式化、不被 UNIX 通过文件系统来读取的特殊块设备文件，由应用程序负责对其进行读/写操作。这种设备由于缺少操作系统这一层，因而 I/O 效率更高。不少数据库都能通过使用裸设备作为存储介质来提高 I/O 效率。

在 UNIX 的/dev 目录下有许多文件，其中有两个大类：字符设备文件和块设备文件。

字符设备文件在进行 I/O 操作时不经过操作系统的缓冲区；而块设备文件用来同外设进行定长的包传输。字符设备文件在与外设进行 I/O 操作时每次只传输一个字符；而对于块设备文件来说，它采用了 Cache 机制，在外设和内存之间一次可以传送一整块数据。裸设备是一种特殊类型的块设备文件。

因为使用裸设备避免了再经过 UNIX 操作系统这一层，数据直接从磁盘传输到 Oracle，所以使用裸设备对于读/写频繁的数据库应用来说，可以极大地提高数据库系统的性能。

1. 裸设备的适用范围

判断是否使用裸设备需要从以下几个方面进行考虑：首先，数据库系统本身已经被比较好地进行了优化；其次，使用 UNIX 命令来判断是否存在磁盘读/写瓶颈。如果决定采用裸设备，则磁盘上要有空闲的分区；否则，需要添加磁盘，或者对原有系统进行重新规划。

如果使用了 Oracle 并行服务器选项，则必须采用裸设备来存放所有的数据文件、控制文件、联机重做日志文件。只有把这些文件都放到裸设备上，才能保证所有 Oracle 实例都可以读取这个数据库中的文件。这是由 UNIX 操作系统的特性决定的。

还有一种情况是，如果想使用异步 I/O，那么在有些 UNIX 操作系统上也必须采用裸设备。这需要参考 UNIX 的相关文档。

2. 裸设备的创建及备份

通常由 root 来创建裸设备，然后再分配给 Oracle 用户。同时还要把它归入 Oracle 用户所在的组里（通常都是 DBA）。

在创建数据文件时指定裸设备和普通文件没有太大的区别，都是在单引号里写上裸设备的详细路径即可。

在裸设备上，不能使用 UNIX 实用程序来进行备份，唯一的办法是使用最基本的 UNIX 命令 dd 来进行备份。比如：`dd if=/dev/raw1 of=/dev/rmt0 bs=16k`。dd 的具体语法可以参考 UNIX 手册，或者在线帮助。也可以先用 dd 把裸设备上的数据文件备份到磁盘上，然后再利用 UNIX 实用程序做进一步处理。

把联机重做日志文件放到裸设备上是一个极好的选择。联机重做日志文件是写操作非常频繁的文件，放到裸设备上非常合适。但是归档日志文件必须放到常规的 UNIX 文件系统上面，或者直接放到磁带上。在提高数据读/写速度和性能方面还有其他的途径，使用 RAID 也是非常有效的办法，尤其是针对那种读/写非常频繁的系统。对 Oracle 进行优化，并且购买更多的磁盘和磁盘控制器来分散 I/O 到不同的磁盘上，也是一种比较有效的方法。

需要注意的是，在 Oracle 中，一旦使用了裸设备，tablespace 就不能设定为 auto_extend，因为在裸设备中不支持文件大小的自动扩展。因此，当数据文件快要写完的时候，请手动在裸设备上添加数据库文件以完成扩容。

10.5.3 对象接口

对象存储系统（Object-Based Storage System）是综合了 NAS 和 SAN 的优点，同时具有 SAN 的高速直接访问和 NAS 的数据共享等优势，提供了高可用性、跨平台性及安全性的数据共享的存储体系结构。

Object 是对象存储的基本单元。每个 Object 都是数据和数据属性集的综合体。数据属性可以根据应用的需求进行设置，包括数据分布、服务质量等。在传统的存储中，块设备要记录每个存储数

据块在设备上的位置。Object 维护自己的属性，从而简化了存储系统的管理任务，增加了灵活性。Object 的大小可以不同，可以包含整个数据结构，如文件、数据库表项等。

1. 对象存储的产生

随着互联网、Web 2.0 的快速发展，Web 应用创建出数百亿的小文件；人们上传海量的照片、视频、音乐，Facebook 每天都会新增数十亿条内容，人们每天都会发送数千亿封电子邮件。据 IDC 统计，未来 10 年间，数据将增长 44 倍，到 2020 年全球数据将增加到 35ZB，其中 80% 是非结构化数据，且大部分是非活跃数据。面对如此庞大的数据量，仅具备 PB 级扩展能力的块存储（SAN）和文件存储（NAS）显得有些无能为力。单个文件系统在最优性能的情况下支持的文件数量通常只在百万级别。人们需要一种全新架构的存储系统，这种存储系统需要具备极高的可扩展性，能够满足人们对存储容量从 TB 到 EB 规模的扩展需求。

2002 年，安然、世界通信等事件的接连爆发导致萨班斯法案推出，对象存储被用于政府法规要求数据长期保存金融服务、健康医疗等行业的数据归档场景，对象存储由此具备了备份归档的基因。

2006 年，Amazon 发布 AWS，S3 服务及其使用的 REST、SOAP 访问接口成为对象存储的事实标准。Amazon S3 成功地为对象存储注入了云服务基因。

2. 对象存储的关键特性与价值

对象存储是一种基于对象的存储设备，具备智能、自我管理能力，通过 Web 服务协议（如 REST、SOAP）实现对象的读/写和存储资源的访问。

对象存储系统包含两种数据描述：容器（Bucket）和对象（Object）。容器和对象都有一个全局唯一的 ID。对象存储采用扁平化结构管理所有数据，用户/应用通过接入码（Access Key）认证后，只需根据 ID 就可以访问容器/对象及相关的数据（Data）、元数据（Metadata）和对象属性（Attribute）。

对象存储数据组织示意图如图 10.3 所示。

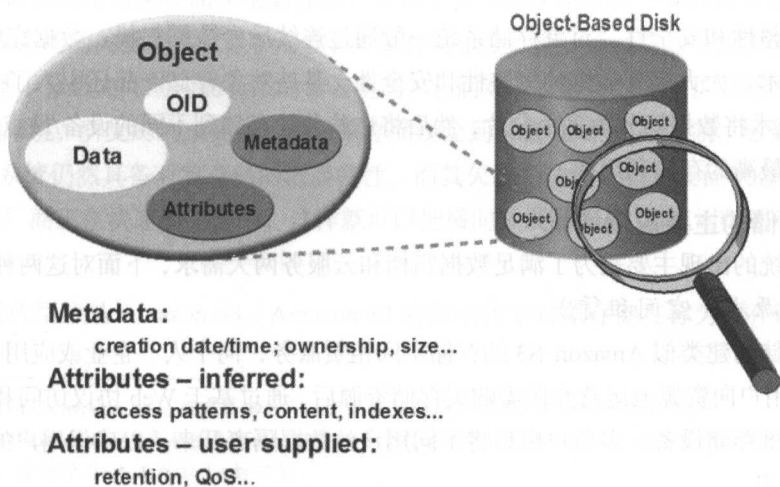


图 10.3

对象存储对外提供更抽象的对象接口，而不是 SCSI 或文件接口。与 SAN 存储以逻辑扇区为单位的较细粒度的固定 I/O（512B~4KB）不同，对象存储 I/O 粒度更有弹性，支持几字节（B）到数万亿字节（TB）范围内的任意对象大小，使得业务可以根据需要灵活地分割数据。对象存储以对象 ID 为基础，扁平化地管理所有对象和桶，根据对象 ID 便可以直接访问数据，解决了 NAS 复杂的目录树结构在海量数据情况下的数据查找耗时较长的问题。这使得对象存储具备极强的扩展性，能够轻松实现单一名字空间（namespace）内支持百亿级文件的存储。

在重复数据删除、绿色节能等特性的基础上，为了更好地满足海量数据存储和公众云服务的需求，对象存储系统还包括如下一些关键特性。

（1）超强的扩展性。扁平化的数据结构允许对象存储容量从 TB 级扩展到 EB 级，管理数十个到百亿个存储对象，支持从数字节（Byte）到数万亿字节（TB）范围内的任意大小对象，解决了文件系统复杂的 iNode 机制带来的扩展性瓶颈，并使得对象存储无须像 SAN 存储那样管理数量庞大的逻辑单元号（LUN）。对象存储系统通常在一个横向扩展（或网格硬件）架构上构建一个全局的命名空间，这使得对象存储非常适合在云计算环境中使用。某些对象存储系统还支持升级、扩容过程中的业务零中断。

（2）基于策略的自动化管理。由于云环境中的数据往往是动态、快速增长的，所以基于策略的自动化将变得非常重要。对象存储支持从应用角度基于业务需求设置对象/容器的属性（元数据）策略，如数据保护级别、保留期限、合规状况、远程复制的份数等。这使得对象存储具备云的自服务特征，同时能有效降低运维管理的成本，使得客户在存储容量从 TB 级增长到 ZB 级时运维管理成本不会随之飙升。

（3）多租户。多租户特性使得对象存储系统可以使用同一种架构、同一套系统为不同用户和应用提供存储服务，并分别为这些用户和应用设置数据保护、数据存储策略，并确保这些数据之间相互隔离。

（4）数据完整性和安全性。对象存储系统一般通过连续后台数据扫描、数据完整性校验、自动化对象修复等技术，大大提高数据的完整性和安全性。某些对象存储产品还引入了一些先进的算法（如擦除码）和技术将数据切分为多个分片，然后将这些分片存储到不同的设备/站点，在确保数据完整性的同时获取最高的存储利用率。

3. 对象存储的主要应用场景

对象存储系统的出现主要是为了满足数据归档和云服务两大需求，下面对这两种场景进行细化。

1）存储资源池（空间租赁）

使用对象存储构建类似 Amazon S3 的存储空间租赁服务，向个人、企业或应用提供按需扩展的弹性存储服务。用户向资源池运营商按需购买存储资源后，通过基于 Web 协议访问和使用存储资源，而无需采购和运维存储设备。多租户模型将不同用户的数据隔离开来，以确保用户的数据安全。

2）网盘应用

在海量存储资源池的基础上，使用图形用户界面（GUI）实现对象存储资源的封装，向用户提供

类似 DropBox 的网盘业务。用户可通过 PC 客户端、手机客户端、Web 页面完成数据的上传、下载、管理与分享。在网盘的帮助下，个人和家庭用户能够实现数据安全、持久保存和不同终端之间的数据同步；企业客户可实现更高效的信息分享、协同办公和非结构化数据管理，同时还可用于实现低成本的 Windows 远程备份，确保企业数据安全。

3) 集中备份

在大型企业或科研机构中，对象存储通过与 CommVault Simpana、Symantec NBU 等主流备份软件结合，可向用户提供更具成本效益、更低 TCO^①的集中备份方案。相对于原有的磁带库或虚拟磁带库等备份方案，重复数据删除特性能够帮助用户用更少的存储设备存储更多的数据，从而减少设备采购，智能管理特性使得备份系统无须即时维护，从而降低了 CAPEX^②和 OPEX^③；分布式并行读/写带来的巨大吞吐量和在线/近线的存储模式有效降低了 RTO^④和 RPO^⑤。

4) 归档和分级存储

对象存储通过与归档软件、分级存储软件结合，将在线系统中的数据无缝归档/分级存储到对象存储，释放在线系统存储资源。对象存储提供几乎可无限扩展的容量及智能管理能力，帮助用户降低海量数据归档的 TCO；对象归档采用主动归档模式，使得归档数据能够被按需访问，而无须长时间地等待和延迟。

4. 与传统存储的比较

基于块的存储系统，磁盘块通过底层存储协议访问，像 SCSI 命令，开销很小，而且没有其他额外的抽象层。这是访问磁盘数据最快的方式，所有高级别的任务，如多用户访问、共享、锁定和安全等通常由操作系统负责。换句话说，基于块的存储关心所有底层的问题，但其他事情都要依靠高层的应用程序实现。而所有的对象存储拥有基于块存储的节点，利用对象存储软件集合提供所有其他的功能。

基于块的存储系统是对象存储系统的补充，而基于文件的存储系统一般被认为是直接的竞争者。横向扩展的 NAS 系统的关键属性就是扩展性，对象存储也是如此，通过增加节点实现水平扩展。但由于 NAS 系统是基于分层文件结构的有限的命名空间，它们对于有着接近无限扩展能力的、具有扁平结构的纯对象存储来说，所受的约束更多，而对象存储仅受到对象 ID 的位数限制。尽管限制多多，但横向扩展的 NAS 系统仍然具备对象存储的诸多特性，而其欠缺的功能，如对表征状态转移 (REST) 协议的支持，各大厂商正在快速地完善中，这样就可以把横向扩展的 NAS 系统划归到对象存储的类别。

5. S3

对象存储最典型的是 Amazon S3。Amazon S3 将数据作为对象存储在称为“存储桶”的资源中。

① TCO (Total Cost of Ownership): 总拥有成本，即从产品采购到后期使用、维护的总的成本。

② CAPEX (Capital Expenditure): 资本性支出，一般是指资金或固定资产、无形资产、递延资产的投入。

③ OPEX=维护费用+营销费用+人工成本 (+折旧)。

④ RTO (Recovery Time Objective): 复原时间目标，是企业可容许服务中断的时间长度。

⑤ RPO (Recovery Point Objective): 复原点目标，是指当服务恢复后，恢复得来的数据所对应的时间点。

用户可以在一个存储桶中尽可能多地存储对象，并写入、读取和删除存储桶中的对象。对象大小最多为 5TB。用户可以控制对存储桶的访问权限（例如，控制谁能在存储桶中创建、删除和检索对象）、查看该存储桶的访问日志及其对象，并选择存储桶存储所在的 AWS 区域以优化延迟性，最大限度地降低成本或满足法规要求。

Amazon S3 为任务关键型和主要数据存储提供了高度持久的存储基础设施。Amazon S3 将数据冗余存储在多个设施中，也存储在每个设施内的多个设备上。为了提高耐久性，Amazon S3 在确认数据已成功存储之前将数据同步存储在多个设施中。此外，Amazon S3 还会在存储或检索数据时对所有的网络流量计算校验和，以检测数据包是否损坏。与传统系统需要费时耗力的数据验证和手工修复方式不同的是，Amazon S3 可以定期执行系统的数据完整性校验，并且内置了自动的自我修复能力。

Amazon S3 的标准存储：以 Amazon S3 服务等级协议作为后盾，以实现可用性；设计目的是在指定年度内为对象提供 99.99999999% 的持久性和 99.99% 的可用性；能够承受两个设施中的数据同时丢失。

10.5.4 块接口^①

块存储，简单来说就是提供了块设备存储的接口。通过向内核注册块设备信息，在 Linux 中通过 `lsblk` 命令可以得到当前主机上的块设备信息列表。

EBS 是 Amazon 提供的块存储服务，通过 EBS，用户可以随时进行增、删、迁移 volume 和快照操作。Amazon EC2 实例可以将根设备数据存储在 Amazon EBS 或者本地实例存储上。在使用 Amazon EBS 时，根设备中的数据将独立于实例的生命周期保留下来，使得在停止实例后仍可以重新启动使用。另外，本地实例存储仅在实例的生命周期内保留。这是启动实例的一种经济方式，因为数据没有存储到根设备中。

Amazon EBS 提供两种类型的卷，即标准卷和预配置 IOPS 卷。它们的性能特点和价格不同，可以根据应用程序的要求和预算定制所需的存储性能。

标准卷可为要求有适度或突发式 I/O 的应用程序提供存储。这些卷平均可以提供大约 100 IOPS，最多可突增至数百 IOPS。标准卷也非常适合用作引导卷，其突发能力可提供快速的实例启动时间（通常为十几秒）。

预配置 IOPS 卷旨在为数据库等 I/O 密集型随机读/写工作负载提供可预计的高性能。在创建一个卷时，利用预置 IOPS 为卷确定 IOPS 速率，随之 Amazon EBS 在该卷的生命周期内提供该速率。Amazon EBS 目前支持每预配置 IOPS 卷最多 4000 IOPS。用户可以将多个条带式卷组合在一起，为应用程序提供每个 Amazon EC2 数千 IOPS 的能力。

EBS 可以在卷连接和使用期间实时拍摄快照。不过，快照只能捕获已写入 Amazon EBS 卷的数

^① 参考 <http://www.cnblogs.com/shishanyuan/p/4637631.html>。

据，不包含应用程序或操作系统已在本地缓存的数据。如果需要确保为实例连接的卷获得一致的快照，则需要先彻底地断开卷连接，再发出快照命令，然后重新连接卷。

EBS 快照目前可以跨区域增量备份，这意味着 EBS 快照时间会大大缩短，同时增加了 EBS 使用的安全性。

1. HDFS

HDFS 是 Hadoop 领域纯软件实现的块接口的存储软件，是 Hadoop 软件栈里面基础的基础。

1) HDFS 原理

HDFS (Hadoop Distributed File System) 是一个分布式文件系统，是 Google 的 GFS 山寨版本。它具有高容错性，同时提供了高吞吐量的数据访问，非常适合在大规模数据集上应用。

- 高吞吐量访问：HDFS 的每个 Block 分布在不同的 Rack 上，在用户访问时，HDFS 自动根据数据存储的位置和服务的负载选择路径最短和负载最小的服务器提供数据访问。由于 Block 在不同的 Rack 上都有备份，所以不再是单数据访问，速度和效率非常快。另外，HDFS 可以并行地从服务器集群中读/写数据，增加了文件读/写的访问带宽。
- 高容错性：系统故障是不可避免的，如何做到故障之后的数据恢复和容错处理是至关重要的。HDFS 通过多种措施来保证数据的可靠性，如数据多份复制并且分布到物理位置的不同服务器上、数据校验功能、后台的连续自检数据一致性功能等。
- 线性扩展：因为 HDFS 的 Block 信息存放在 NameNode 上，文件的 Block 分布在 DataNode 上，因此，当需要扩充的时候，仅仅需要添加 DataNode 的数量，系统即可在不停止服务的前提下进行扩充，而不需要人工干预。

2) HDFS 架构

HDFS 的架构如图 10.4 所示。

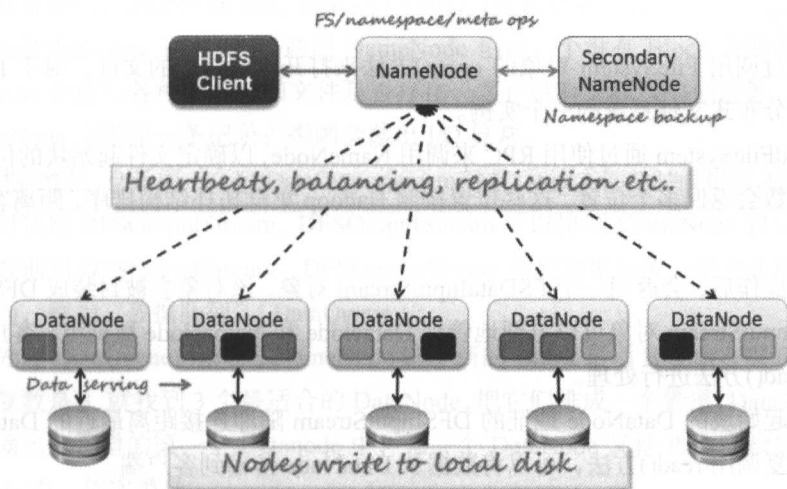


图 10.4

HDFS 是 Master-Slave 结构，分为 NameNode、SecondaryNameNode 和 DataNode 三种角色。

- NameNode: 在 Hadoop 1.X 中只有一个 Master 节点，管理 HDFS 的命名空间和数据块映射信息、配置副本策略和处理客户端请求。
- SecondaryNameNode: 辅助 NameNode，分担 NameNode 的工作，定期合并 fsimage 和 fsedit 并推送给 NameNode，紧急情况下可辅助恢复 NameNode。
- DataNode: Slave 节点，实际存储数据，执行数据块的读/写，并汇报存储信息给 NameNode。

3) HDFS 读操作

HDFS 读操作示意图如图 10.5 所示。

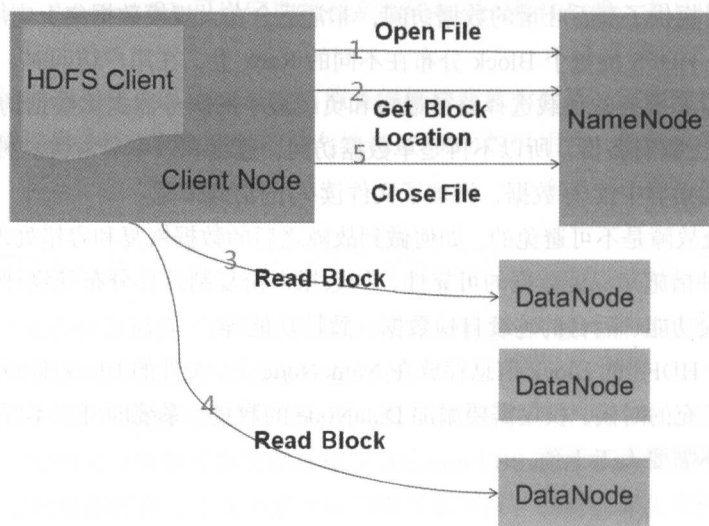


图 10.5

具体流程如下：

- 客户端通过调用 `FileSystem` 对象的 `open()` 方法来打开希望读取的文件。对于 HDFS 来说，这个对象是分布式文件系统的一个实例。
- `DistributedFileSystem` 通过使用 RPC 来调用 NameNode，以确定文件起始块的位置。同一 Block 按照重复数会返回多个位置，这些位置按照 Hadoop 集群拓扑结构排序，距离客户端近的排在前面。
- 经过上述操作后，会返回一个 `FSDDataInputStream` 对象，该对象会被封装成 `DFSInputStream` 对象。`DFSInputStream` 对象可以方便地管理 DataNode 和 NameNode 数据流，客户端对这个输入流调用 `read()` 方法进行处理。
- 存储文件起始块的 DataNode 地址的 `DFSInputStream` 随即连接距离最近的 DataNode，通过对数据流反复调用 `read()` 方法，可以将数据从 DataNode 传输到客户端。
- 到达块的末端时，`DFSInputStream` 会关闭与该 DataNode 的连接，然后寻找下一个块的最

佳 DataNode。这些操作对客户端来说是透明的，从客户端的角度来看只是读一个持续不断的流。

- 一旦客户端完成读取，就会调用 close() 方法关闭文件读取。

4) HDFS 写操作

HDFS 写操作示意图如图 10.6 所示。

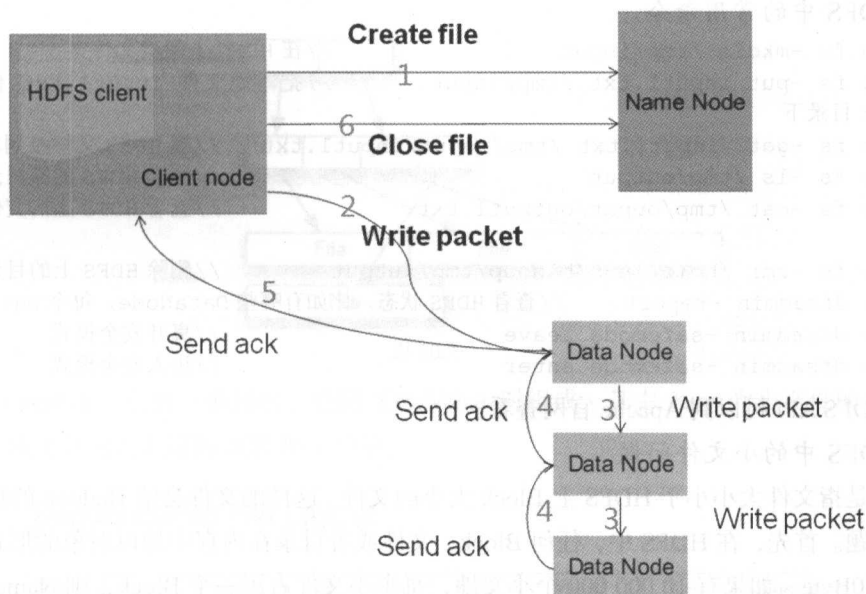


图 10.6

具体流程如下：

- 客户端通过调用 DistributedFileSystem 的 create() 方法创建新文件。
- DistributedFileSystem 通过 RPC 调用 NameNode 创建一个没有 Block 关联的新文件，创建前 NameNode 会进行各种校验，如文件是否存在、客户端有无权限去创建等。如果校验通过，则 NameNode 会创建一条记录；否则会抛出 I/O 异常。
- 经过上述操作后，会返回一个 FSDataOutputStream 对象。和读文件的时候相似，FSDataOutputStream 对象被封装成 DFSOutputStream，DFSOutputStream 可以协调 NameNode 和 DataNode。客户端开始写数据到 DFSOutputStream，DFSOutputStream 会把数据切成一个个小的数据包，并写入内部队列，称为“数据队列”（DataQueue）。
- 接下来处理 DataQueue，先询问 NameNode 这个新的 Block 最适合存储在哪几个 DataNode 里，如果重复数是 3，就找到 3 个最适合的 DataNode，把它们排成一个管道。DataStreamer 把 Packet 按队列输出到管道的第一个 Datanode 中，第一个 DataNode 又把 Packet 输出到管道的第二个 DataNode 中，依次类推。

- DFSOutputStream 还有一个队列叫 AckQuene,也是由 Packet 组成的。当管道中的所有 DataNode 都表示已经收到响应的时候, AkcQuene 才会把对应的 Packet 包移除。
- 客户端完成写数据后, 调用 close()方法关闭写入流。
- DataStreamer 把剩余的包都 flush 到 Pipeline 里, 然后等待 Ack 信息。在收到最后一个 Ack 信息后, 通知 NameNode 把文件标识为已完成。

5) HDFS 中的常用命令

```
hadoop fs -mkdir /tmp/input //在 HDFS 上新建文件夹
hadoop fs -put input1.txt /tmp/input //把本地文件 input1.txt 传到 HDFS 的
/tmp/input 目录下
hadoop fs -get input1.txt /tmp/input/input1.txt //把 HDFS 文件拉到本地
hadoop fs -ls /tmp/output //列出 HDFS 的某目录
hadoop fs -cat /tmp/ouput/output1.txt //查看 HDFS 上的文件

hadoop fs -rmr /home/less/hadoop/tmp/output //删除 HDFS 上的目录
hadoop dfsadmin -report //查看 HDFS 状态, 比如有哪些 DataNode, 每个 DataNode 的情况
hadoop dfsadmin -safemode leave //离开安全模式
hadoop dfsadmin -safemode enter //进入安全模式
```

相关 HDFS API 可以到 Apache 官网查看。

6) HDFS 中的小文件问题^①

小文件是指文件大小小于 HDFS 上 Block 大小的文件。这样的文件会给 Hadoop 的扩展性和性能带来严重问题。首先, 在 HDFS 中, 任何 Block、文件或者目录在内存中均以对象的形式存储, 每个对象约占 150Byte。如果有 10 000 000 个小文件, 每个小文件占用一个 Block, 则 NameNode 大约需要 2GB 空间。如果存储 1 亿个小文件, 则 NameNode 需要约 20GB 空间。这样一来, NameNode 的内存容量严重制约了集群的扩展。其次, 访问大量小文件的速度远远小于访问几个大文件。HDFS 最初是为流式访问大文件而开发的, 如果访问大量小文件, 则需要不断地从一个 DataNode 跳到另一个 DataNode, 严重影响了性能, 导致最后处理大量小文件的速度远远小于处理同等大小的大文件的速度。每个小文件要占用一个 Slot, 而 Task 启动将耗费大量时间, 从而导致大部分时间都耗费在启动和释放 Task 上。

要想解决小文件的问题, 就要想办法减少文件数量, 降低 NameNode 的压力。通常有两种解决方法: 一种是用户程序合并, 另一种是从机制上支持小文件的合并。

① 用户程序合并^②

Hadoop 自身提供了三种解决方案: HadoopArchive、SequenceFile 和 CombineFileInputFormat。

(1) HadoopArchive: 归档为 bar.har 文件, 该文件的内部结构如图 10.7 所示。

创建存档文件的问题:

- 存档文件的源文件目录及源文件都不会自动删除, 需要手动删除。

① 参考 <http://www.open-open.com/lib/view/open1330605869374.html>。

② 参考 <http://blog.cloudera.com/blog/2009/02/the-small-files-problem/>。

- 存档的过程实际是一个 MapReduce 过程，所以需要 Hadoop 的 MapReduce 的支持。
- 存档文件本身不支持压缩。
- 存档文件一旦创建便不可修改，要想从中删除或者增加文件，必须重新建立存档文件。
- 创建存档文件会创建原始文件的副本，所以至少需要有与存档文件容量相同的磁盘空间。

HAR File Layout

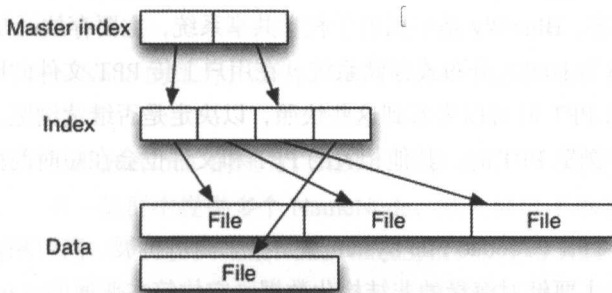
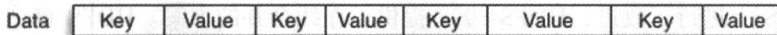


图 10.7

(2) SequenceFile: 它由一系列的二进制 Key/Value 对组成，其中 Key 为小文件的名字，Value 为文件内容。该文件的内部结构如图 10.8 所示。

SequenceFile File Layout



MapFile File Layout

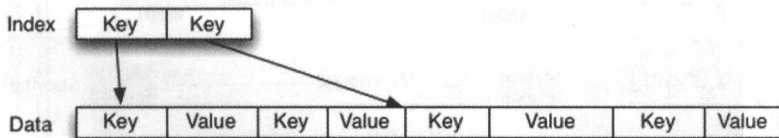


图 10.8

创建 SequenceFile 的过程可以使用 MapReduce 的工作方式完成。对于 Index，需要改进查找算法，对小文件的存取都比较自由，也不限制用户和文件的多少。但是该方法不能使用 append 方法，所以适合一次性写入大量小文件的场景。

(3) CombineFileInputFormat: 它是一种新的 Inputformat，用于将多个文件合并成一个单独的 Split。另外，它会考虑数据的存储位置。

② 通用合并方法

业界针对数据的不同特征，有一些合并优化的方法，可以降低文件数量、提高存储性能。

(1) WebGIS 解决方案。在地理信息系统中，为了方便传输，通常将数据切分为 KB 大小的文件存储在分布式文件系统中。论文结合 WebGIS 数据的相关特征，将相邻地理位置的小文件合并成一个大的文件，并为这些文件构建索引。论文中将小于 16MB 的文件当作小文件进行合并处理，将其合并成 64MB 的 Block 并构建索引。

(2) BlueSky 解决方案。BlueSky 是中国电子教学共享系统，主要存放的是教学所用的 PPT 文件和视频文件，存放的载体为 HDFS 分布式存储系统。在用户上传 PPT 文件的同时，系统还会存储一些文件的快照。用户请求 PPT 时可以先看到这些快照，以决定是否继续浏览。用户对文件的请求具有很强的关联性。当用户浏览 PPT 时，其他相关的 PPT 和文件也会在短时间内被访问，因而文件的访问具有相关性和本地性。

(3) TFS 解决方案。TFS (Taobao File System) 是一个高可扩展、高可用、高性能、面向互联网服务的分布式文件系统，主要针对海量的非结构化数据，它构筑在普通的 Linux 机器集群上，可为外部提供高可靠和高并发的存储访问。TFS 为淘宝提供海量小文件存储，通常文件大小不超过 1MB，满足了淘宝对小文件存储的需求，被广泛应用在淘宝的各项应用中。它采用了 HA 架构和平滑扩容，保证了整个文件系统的可用性和扩展性。同时扁平化的数据组织结构可将文件名映射到文件的物理地址，简化了文件的访问流程，一定程度上为 TFS 提供了良好的读/写性能。

③ 小文件社区最新改进 HDFS-8998^①

社区在 HDFS 上进行了改进，HDFS-8998 提供了在线合并的方案。HDFS 自动启动一个服务，将小文件合并成大文件。其主要架构如图 10.9 所示。

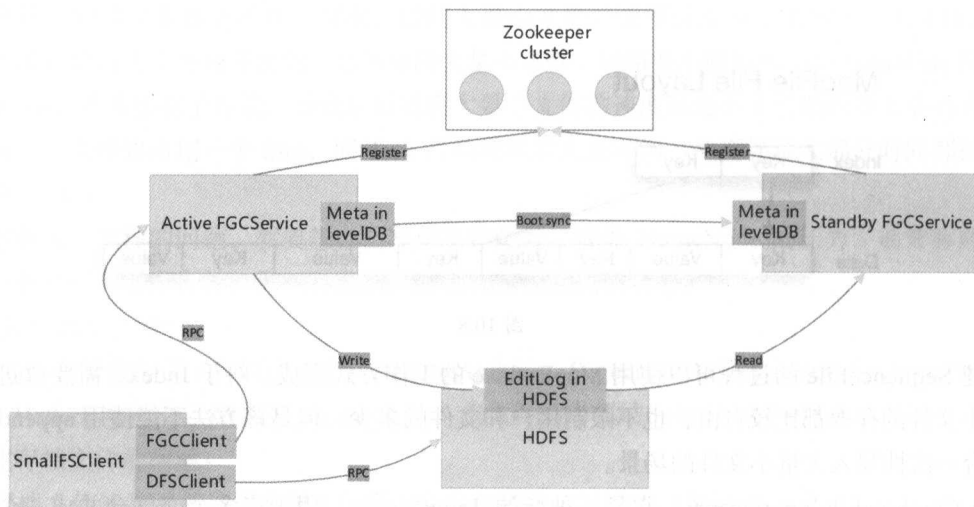


图 10.9

^① 参考 <https://issues.apache.org/jira/browse/HDFS-8998>。

相比原生 HDFS，新增一个 FGCServer 的后台服务，服务本身支持 HA。元数据存储在 levelDB 中，文件和日志都存储在 HDFS 本身。

后台服务自动搜索小文件，合并符合规则的小文件到大文件。小文件合并成大文件需要记录小文件在大文件里面的大小、偏移位置、对应关系等信息，这些元数据存储在 levelDB 中。

因为合并后原始文件的存储位置发生了变更，所以原 HDFS 的读/写等接口的流程也发生了变更。比如，要读取一个文件，需要先到 FGCService 中获取小文件元数据，然后再到 HDFS 中获取对应的文件。

通过合并，减轻了 NameNode 的压力，增大了 HDFS 单个 NameNode 支持的文件个数。

④ HDFS Federation^①

前面讲到通过合并降低 NameNode 的压力，HDFS 开源了另一个方案 HDFS Federation，通过 NameNode 的 Federation，单一集群中提供多个 NameNode，从而降低了单个 NameNode 的压力。

HDFS Federation 使用了多个独立的 NameNode/NameSpace 来使得 HDFS 的命名服务能够水平扩展。在 HDFS Federation 中，NameNode 之间相互独立且不需要相互协调，提供了命名空间和块管理功能；DataNode 被所有的 NameNode 用作公共存储块的地方，每个 DataNode 都会向所在集群中所有的 NameNode 注册，并且会周期性地发送心跳和块信息报告，同时处理来自 NameNode 的指令，如图 10.10 所示。

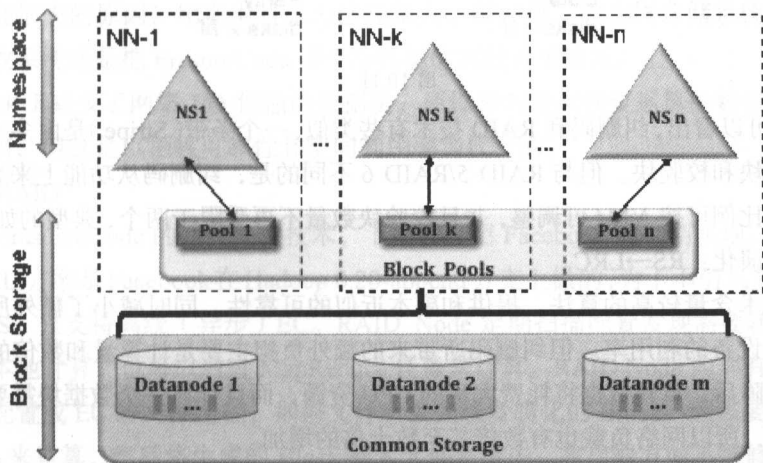


图 10.10

7) HDFS RAID 技术

HDFS 为了提高可靠性，采用了“暴力”的三副本技术，这就带来了存储成本的问题。业界一直在探索如何降低存储成本。传统的 RAID 技术就被很自然地借鉴过来与 HDFS 相结合。通过结合 RAID 的 ErasureCode 技术，存储成本从原来的 HDFS 默认的 3 倍降低到 1.4 倍。

① 参考 <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/Federation.html> 和 <http://blog.csdn.net/strongerbit/article/details/7013221>。

下面先介绍一下 ErasureCode 技术，再讲解一下目前 Hadoop 社区的两种主流实现。

① ErasureCode 技术^①

(1) ErasureCode 原理。

纠删码常见的有三类：Reed-Solomon 类、级联低密度纠删码和数字喷泉码。这里只简单介绍一下目前广泛应用的 Reed-Solomon 类纠删码。

从纠删码的基本形态来看，它是 N 个数据+ M 个校验的结构，其中数据和校验的 N 和 M 值都能够按照一定的规则设定。在 $1 \sim M$ 个数据块（数据或校验均可）损坏的情况下，整体数据仍然可以通过计算剩余数据块上的数据得出，整体数据不会丢失，存储仍然可用。

纠删码的结构示意图如图 10.11 所示。

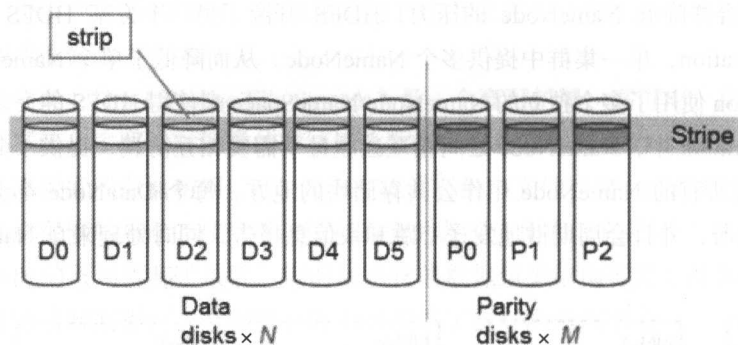


图 10.11

从图 10.11 中可以看出，纠删码和 RAID 技术有些类似，一个条带 (Stripe) 是由多个数据块 (Strip) 构成的，分为数据块和校验块。但与 RAID 5/RAID 6 不同的是，纠删码从功能上来看最大的区分特点是校验和数据的比例可按 $N+M$ 可调整，并且校验块数量不再受限于两个，典型的如 $12+4$ 、 $6+3$ 等。

(2) 纠删码的演化：RS→LRC。

纠删码通过技术含量较高的算法，提供和副本近似的可靠性，同时减小了额外所需冗余设备的数量，提高了存储设备的利用率。但纠删码所带来的额外负担主要是计算量和数倍的网络负载。尤其是在出现硬盘故障后，重建数据将耗费大量的 CPU 资源，而且计算一个数据块需要通过网络读出 N 倍的数据并传输，所以网络负载也有数倍甚至数十倍的增加。

从整体来看，若采用纠删码技术，则能够得到希望的容错能力和存储资源利用率，但是需要接受一定的数据重建代价，二者之间需要进行平衡。

那么，有没有优化改善的空间呢？答案是“有”。

如果仔细分析故障出现的原因，则会很容易发现两个特征。

- 特征一：所有的故障都将导致同样的重建代价，无论是一个盘还是 M 个盘。
- 特征二：单个磁盘出现故障的概率远远大于多个磁盘同时出现故障的概率，通常在 90% 以上。

① 参考 http://blog.sina.com.cn/s/blog_57f61b490102viq9.html

因此,优化的思路自然聚集到更容易出现的单个磁盘故障上来。如何更有效地处理这种概率较大的事件呢?最佳的解决方案是分组,把单个磁盘故障的影响范围缩小到各个组内部,当磁盘出现故障时,在该组内部解决,在恢复过程中读组内更少的盘,跑更少的网络流量,从而减小对全局的影响。

LRC (Locally Repairable Codes) 意为局部校验编码,其核心思想为:将校验块 (Parity Block) 分为全局校验块 (Global Parity) 和局部校验块 (Local Reconstruction Parity),在故障恢复时分组计算。

以微软 Azure 的云存储 (Windows Azure Storage) 实现为例,它采用 LRC(12,2,2) 编码,12 个数据块为一组编码,并进一步将这 12 个数据块平均分为两个本地组,每个本地组包括 6 个数据块,并分别计算出一个 Local Parity,之后把所有 12 个数据块计算出两个全局校验块。

当发生任何一个数据块错误时,只需将本地组内的数据和校验块用于计算,即可恢复出原始数据。而恢复代价 (通过网络传输的数据块数量) 就由传统 RS(12,4) 编码的 12 变为 6,恢复过程的网络 I/O 开销减半,同时空间冗余率保持不变,仍为 $(12+2+2)/12=1.33$ 。

(3) ErasureCode 总结。

相对于副本而言,纠删码 (ErasureCode) 的编码技术无疑对存储空间利用率带来很大提升,但由于引入额外的编码、解码运算,对分布式系统的计算能力和网络都有一定的额外要求。简单地理解就是硬件性能要升级,网络环境也要升级,升级的代价在现阶段还是一笔不小的预算。

而由于性能损失的原因,用在本身压力已经很大、很“热”的在线存储系统明显不是很合适,所以目前大多数系统还是把 ErasureCode 用于冷数据的离线处理阶段。

LRC 编码由于减少了网络 I/O 传输的数据量,所以参与数据恢复运算的数据量和重建时间基本上能够缩短一倍,但这是以牺牲可靠性和空间利用率为代价的。

② HDFS RAID^①

前面讲到 ErasureCode 的两种典型技术,下面先介绍 Facebook 的早期实现。

HDFS RAID 方案是 Facebook 在 Hadoop 0.20-append 分支上做的,为了不引入复杂度,基于 HDFS,没有修改 HDFS,只支持离线 (异步) EC。RAID Node 定期扫描配置发现需要转换成 EC 的文件,转换过程可以本地计算,也可以通过 MapReduce 任务来计算。RAID Node 内部有一个 BlockFixer 线程定期检查被配置成 EC 的文件路径,如果文件有丢失或者腐化的 Block,则需要本地重算或者通过 MapReduce Job 来重算,然后将生成的 Block 插入文件系统中。客户端方面不需修改任何代码,只需修改配置,告诉它使用的文件系统是 DistributedRaidFileSystem。它封装了 DFS Client,截获 DFS Client 的请求,当 DFS Client 抛出 ChecksumException 或者 BlockMissingException 时,DRFS 捕获这些异常,定位到相应的 Parity File,然后重新计算出丢失的 Block,随后返回给客户端。

(1) HDFS 整体结构。

HDFS RAID 的实现 (Facebook 的实现) 主要是在现有的 HDFS 上增加了一个包装 Contrib,如

^① 参考 <http://jiangbo.me/blog/2012/12/21/hdfs-raid/>。

图 10.12 所示。之所以不在 HDFS 上直接修改，原设计者的解释是“HDFS 的核心代码已经够复杂了，不想让它更复杂”。

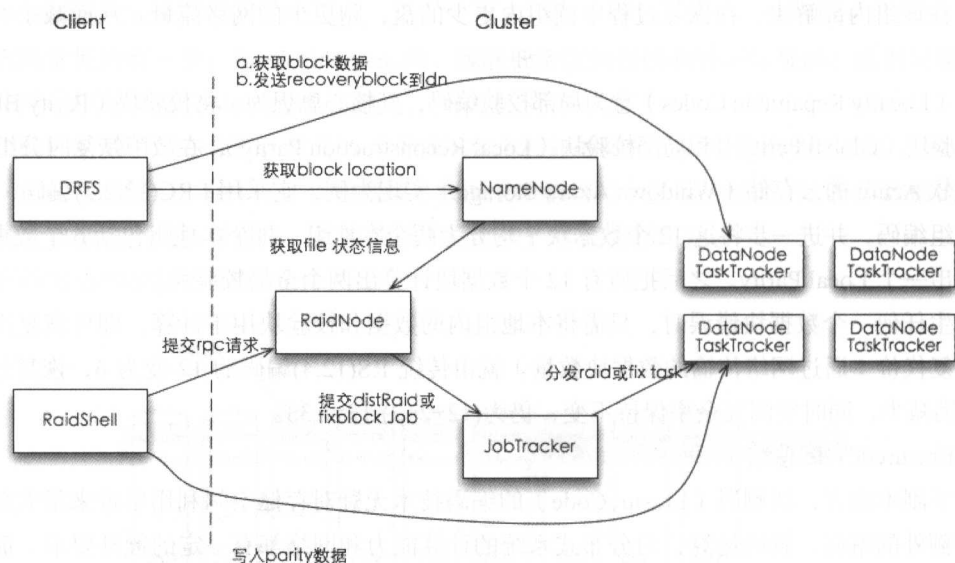


图 10.12

a. 从使用的角度看，HDFS RAID 的使用场景主要有两个：RAID 数据管理和 RAID 数据读取。

b. RAID Node 结构（Server 端）。RAID Node 是 HDFS RAID 中除 NameNode 和 JobTracker 之外的第三个 MasterNode，主要用于接收 Client 端的 RPC 请求和调度各守护线程完成数据的 RAID 化和数据修复、Parity 文件删除等操作。

（2）RAID 和 unRAID 流程详解。

a. 数据 RAID 化。

文件数据的 RAID 化有两种场景：一种是通过 RAID Shell 执行 raidFile 命令触发；另一种是 TiggerMonitor 线程周期性地扫描 Policy，根据新的配置信息进行相应的 RAID 化。

b. 损坏数据的恢复。

RAID 数据的修复同样也有多个触发场景：

- Client 端使用 DRFS 读取数据发生数据丢失或损坏延长。
- RAID Node 上的 BlockIntegrityMonitor 周期性地获取 Block 数据发现数据异常时。
- 通过 RAID Shell 执行 fixblock 时。

Block 读取时修复损坏数据的流程如下：

- 在 Client 端配置 DRFS 并使用 DFS 作为内置 FS 的情况下，当通过 FS.open 获取文件 InputStream 时，返回一个 ExtFSDataInputStream 实例。
- 通过该 InputStream 读取数据时，首先通过内置的 DFS 读取响应的 Block，正常情况下返回需要的数据。

- 当内置的 DFS 读取 Block 时, 若跑出 `CorruptionException` 或 `DecomissionException`, 则会被 `ExtFSDataInputStream` 捕获。通过调用 `RaidNode.unRaidCorruptionBlock()` 方法来获取一个恢复的 Block, 并从该 Block 读取数据。

BlockIntegrityMonitor 线程修复: RAID Node 上的 BlockIntegrityMonitor 线程会通过文件检查工具检查系统中冲突或失效的数据, 然后通过 BlockCopier 和 BlockFixer 线程周期性地对这些数据进行修复。在 Local 模式下, 修复过程在 RaidNode 上执行; 而在 Dist 模式下, 修复过程通过提交 Job 的方式交由集群完成。

③ HDFS-7285^①

(1) 集成 EC 在 HDFS 中。

HDFS-7285 和 HDFS RAID 不同, 它将编码过程集成到 HDFS 内部, 需要对整个 HDFS 内部实现进行改造, 包括 DataNode、NameNode 及 DFS Client。该方案同时支持在线和离线 EC。

(2) 小结: HDFS-7285 比 HDFS RAID 更全面、性能更好, 在线 EC 可以立即降低存储成本。

④ Ozone^②

前面讲到, HDFS 是应用级的块存储, 为了支持非文件性质的系统数据, Hortonworks 改进了 HDFS, 将块存储和对象存储融合, 提出 Ozone, 将 HDFS 从文件系统扩展到更复杂的企业应用。

过去, HDFS 架构将元数据管理与数据存储分离成两个相互独立的层。文件数据存储在有上千个存储服务器(节点)的存储层, 而元数据存储的文件元数据层。HDFS 的这种分离方式使得应用直接从存储磁盘读/写数据时能够获得很高的吞吐量扩展空间。

Ozone 使得 HDFS 块存储层能够进一步支持非文件性质的系统数据, 而 HDFS 的文件块架构也将支持存储键值和对象。与 HDFS 的名称空间元数据类似, Ozone 的元数据系统也基于块存储层, 但是 Ozone 的元数据将被动态分配, 支持大量的 Bucket Space。

Hortonworks 认为, HDFS 将自然进化成一个完整的企业大数据存储系统, 而 Ozone 也将以 Apache 项目 (HDFS-7240) 的方式开源。

⑤ vSAN^③

HDFS 是根据 Google 的论文启发而发明的块存储, 但是 HDFS 应该理解为应用级的存储, 程序完全构建在 OS 普通用户态空间里, 存储的可靠性靠“暴力”的三副本来保证。访问 HDFS 需要使用专门的接口, 因此 HDFS 只适合在 Hadoop 领域使用简单的块存储软件, 并没有存储很多企业特性, 如快照、重复数据消除等。

传统的存储都构筑在专门的 SAN 硬件上, 通信接口也采用 SAS 等存储专有接口, 专有服务器带来的成本高、扩展性差等缺陷越来越不被客户所接受。随着分布式技术的发展, 一些专门做存储的

① 参考 <http://www.cnblogs.com/foxmailed/p/4282130.html>。

② 参考 <https://issues.apache.org/jira/browse/HDFS-7240>。

③ VSAN 博客 <http://vsdsrevolution.blog.51cto.com/>。

厂商自然而然地想到用通用服务器来做存储，于是就有了一系列解决方案，最有名的是 VMware 的 Virtual SAN，如图 10.13 所示。

Virtual SAN for vSphere environments

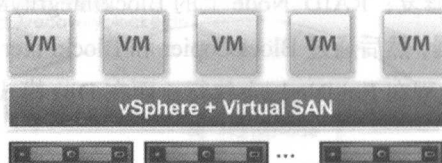


图 10.13

10.5.5 融合是趋势

前面讲述了文件接口、裸设备、对象接口、块接口等各种存储形态和软件技术架构。各种接口的存储都有其产生的原因和生存的空间。在未来很长一段时间内，这些接口会并存。在一个企业/组织里面有不同的数据，大数据的核心价值之一是将这些数据融合起来发挥更大的作用。因此，数据驱动存储技术未来会逐渐走向融合。

10.6 存储加速技术

数据存储下来就是为了应用。本节重点探讨业界在数据组织、索引等技术上的发展和通用思路。

10.6.1 数据组织技术

笔者认为，数据组织是关键。要高效地使用数据，就必须要有组织，因此业界对数据的结构化组织有很多探索。下面介绍几种典型的组织形式，以及它们的优缺点。

1. Cube 技术

1) 概念

OLAP 的目标是满足决策支持或者满足在多维环境下特定的查询和报表需求，它的技术核心是“维”这个概念。“维”（Dimension）是人们观察客观世界的角度，是一种高层次的类型划分。“维”一般包含着层次关系，这种层次关系有时会相当复杂。通过把一个实体的多项重要属性定义为多个维，使用户能对不同维上的数据进行比较。因此，OLAP 也可以说是多维数据分析工具的集合。OLAP 的基本多维分析操作有钻取、切片和切块，以及旋转等。

（1）钻取是为了改变维的层次，变换分析的粒度。它包括向上钻取（rollup）和向下钻取（drilldown）。rollup 是在某一维上将低层次的细节数据概括到高层次的汇总数据，或者减少维数；而 drilldown 则相反，它从汇总数据深入到细节数据进行观察，或增加维数。

（2）切片和切块是在一部分维上选定值后，观察数据在剩余维上的分布。如果剩余的维只有两

个,则是切片;如果有三个,则是切块。

(3) 旋转是为了变换维的方向,即在表格中重新安排维的放置(如行列互换)。

OLAP有多种实现方法,根据存储数据的方式不同可以分为ROLAP、MOLAP、HOLAP。ROLAP表示基于关系型数据库的OLAP实现(Relational OLAP)。以关系型数据库为核心,以关系型结构进行多维数据的表示和存储。ROLAP将多维数据库的多维结构划分为两类表:一类是事实表,用来存储数据和维关键字;另一类是维表,即对每个维至少使用一张表来存放维的层次、成员类别等维的描述信息。维表和事实表通过主关键字和外关键字联系在一起,形成了“星形模式”。对于层次复杂的维,为避免冗余数据占用过大的存储空间,可以使用多张表来描述,这种星形模式的扩展称为“雪花模式”。其特点是将细节数据保留在关系型数据库的事实表中,聚合后的数据也保存在关系型数据库中。这种方式查询效率最低,不推荐使用。

MOLAP表示基于多维数据组织的OLAP实现(Multidimensional OLAP)。以多维数据组织方式为核心,也就是说,MOLAP使用多维数组存储数据。多维数据在存储中将形成“立方块(Cube)”的结构,在MOLAP中对“立方块”的“旋转”、“切块”、“切片”是产生多维数据报表的主要技术。其特点是将细节数据和聚合后的数据均保存在Cube中,所以以空间换效率,查询时效率高,但生成Cube时需要大量的时间和空间。

HOLAP表示基于混合数据组织的OLAP实现(Hybrid OLAP)。如低层是关系型的,高层是多维矩阵型的。这种方式具有更好的灵活性。其特点是将细节数据保留在关系型数据库的事实表中,但是聚合后的数据保存在Cube中,聚合时需要比ROLAP更多的时间,查询效率比ROLAP高,但低于MOLAP。

Cube是典型的以空间换时间的技术。为了提高查询效率,提前以各种维度把数据组织好,如图10.14所示。

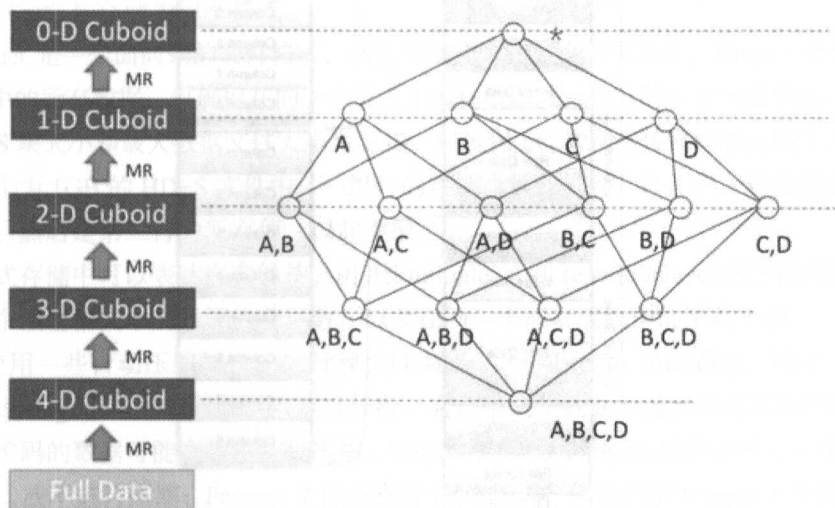


图 10.14

2) Kylin

Apache Kylin 是由 eBay 开源的分布式分析引擎，提供基于 Hadoop 的 SQL 查询接口及多维分析（OLAP）能力，以支持超大规模数据。Kylin 的架构如图 10.15 所示。

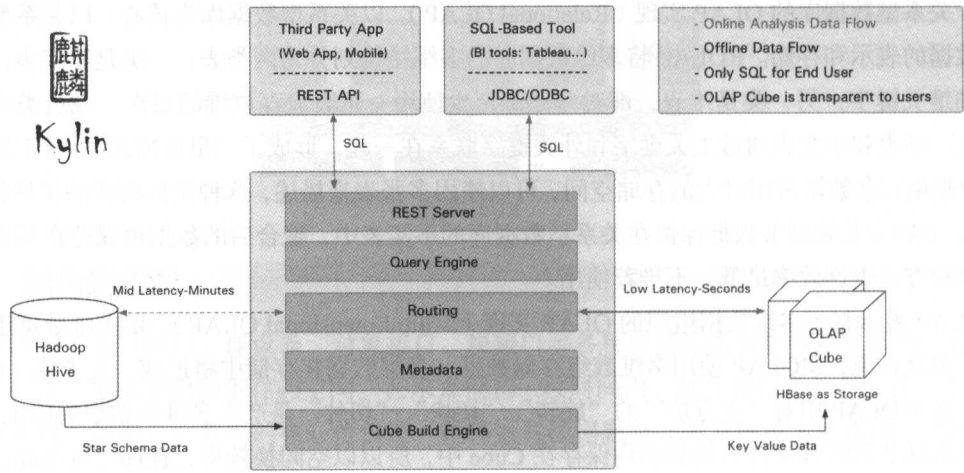


图 10.15

3) ORCFile

ORCFile (Optimized Row Columnar) 是 Hive 0.11 版本中引入的新的存储格式，是对之前的 RCFile 存储格式的优化，是 HortonWorks 开源的。ORCFile 的存储格式如图 10.16 所示。

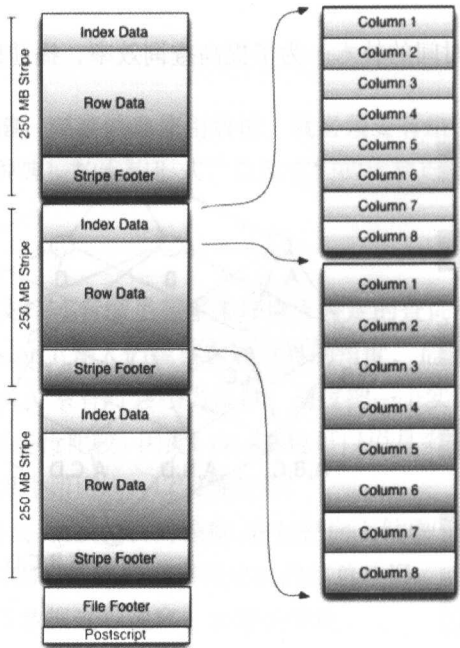


图 10.16

可以看到,每个 ORC 文件由一个或多个 Stripe 组成,每个 Stripe 的大小为 250MB,这个 Stripe 实际上相当于 RCFile 里的 RowGroup,不过大小由 4MB 扩展到 250MB,能够提升顺序读的吞吐率。

每个 Stripe 都包含 IndexData、RowData 及 StripeFooter 三部分。StripeFooter 包含流位置的目录;RowData 在表扫描的时候会用到;IndexData 包含每列的最大值和最小值及每列所在的行。行索引里提供了偏移量,它可以跳到正确的压缩块位置。

通过行索引,可以在 Stripe 快速读取的过程中跳过很多行。在默认情况下,最多可以跳过 10 000 行。

因为可以通过过滤预测跳过很多行,因而可以在表的 SecondaryKeys 进行排序,从而可以大幅度地减少执行时间。

每个文件都有一个 FileFooter,里面存放的是每个 Stripe 的行数、每个 Column 的数据类型等信息;每个文件的尾部是一个 PostScript,里面记录了整个文件的压缩类型及 FileFooter 的长度信息等。在读取文件时,会跳到文件尾部读 PostScript,从里面解析到 FileFooter 长度;再读 FileFooter,从里面解析到各个 Stripe 信息;再读各个 Stripe,即从后往前读。

ORCFile 的主要特点如下:

- 混合存储结构,先按行存储,一组行数据叫 Stripes,Stripes 内部按列存储。
- 支持各种复杂的数据类型。
- 在文件中存储了一些轻量级的索引数据。
- 基于数据类型的块模式压缩: Integer 类型的列用行程长度编码 (Run-Length Encoding, RLE); String 类型的列用字典编码。

4) Parquet

开源项目 Parquet 是 Hadoop 上一种支持列式存储的文件格式,起初只是 Twitter 和 Coudera 在合作开发,发展到现在已经有包括 Criteo 公司在内的许多其他贡献者了。Parquet 用 Dremel 的论文中描述的方式,把嵌套结构存储为扁平格式。

尽管 Parquet 是一个面向列的文件格式,但不要期望每列一个数据文件。Parquet 在同一个数据文件中保存一行中的所有数据,以确保在同一个节点上进行处理时,一行的所有列都可用。Parquet 所做的是设置 HDFS 块大小和最大数据文件大小为 1GB,以确保 I/O 和网络传输请求适用于大批量数据。

在一个大小为 1GB 的 HDFS 文件中,一组行的数据会重新排列,以便第一行的所有值被重组为一个连续的块;然后是第二行的所有值,以此类推。

为了在列式存储中可以表达嵌套结构,用 definitionlevel 和 repetitionlevel 两个值来描述,分别表达某个值在整个嵌套格式中的最深嵌套层数,以及在同一个嵌套层级中的第几个值。

Parquet 使用一些自动压缩技术,如行程长度编码 (Run-Length Encoding, RLE) 和字典编码 (Dictionary Encoding),基于实际数据值进行分析。通过字典使数据值被编码成紧凑的格式,同时使用压缩算法,编码的数据可能会被进一步压缩。Impala 创建的 Parquet 数据文件可以使用 Snappy、Gzip 进行压缩,或不进行压缩;Parquet 文件还支持 LZO 压缩,但是目前 Impala 不支持 LZO 压缩的 Parquet 文件。

除了应用到整个数据文件的 Snappy 或 Gzip 压缩外, RLE 和字段编码是 Impala 自动应用到 Parquet 数据值群体的压缩技术。

综合来看, ORCFile 和 Parquet 本质上都是列式存储, 大同小异。Parquet 的主要特点是支持嵌套格式, ORCFile 的主要特点是 Strips 中有轻量级的 IndexData, 所以这两种数据存储格式完全可以相互借鉴融合。另外, 列式存储不是 Hadoop 首创的, 而是从传统数据库中发展而来的。

5) Google Mesa 数据模型^①

Google 发表了一篇有关大数据系统的论文, 讨论了一个名为 Mesa 的数据仓库系统, 它能处理近实时数据, 即使在整个数据中心断线后还能正常工作。

Mesa 是一个高度可扩展的分析数据仓库系统, 能存储与 Google 广告业务有关的关键测量数据。Mesa 能满足复杂和具有挑战性的用户与系统需求, 包括近实时数据提取和查询, 同时在海量数据和查询量中保持高可用性、可靠性、容错率和扩展性。Mesa 每秒能处理数百万行更新, 每天能进行数十亿次查询, 抓取数万亿行数据。Mesa 能进行跨数据中心复制, 即使在整个数据中心发生故障时, 也能以低延迟返回一致和可重复的查询结果。

针对数分钟更新吞吐量、跨数据中心等严苛需求, 已有的商业数据仓库系统(处理周期往往以天和周来计算)和 Google 的解决方案包括 BigTable、MegaStore、Spanner 和 F1 都无法满足要求。BigTable 无法提供必要的原子性, MegaStore、Spanner 和 F1 无法满足峰值更新需求。此外, Google 自己开发的 Tenzing、Dremel, 以及 Twitter 开发的 Scribe、LinkedIn 的 Avatara、Facebook 的 Hive 及 Hadoop DB 等 Web 规模数据仓库处理的都是批量负载。

Mesa 的主要特点如下:

- (1) 近实时地更新吞吐量。支持持续更新, 每秒支持数百万行更新。
- (2) 同时支持低时延查询性能和批量大量查询。99%的查询在几百毫秒之内返回。
- (3) 跨数据中心备份。

HDFS 最早设定的是数据不更新, 只增量叠加。传统数据仓库(如 Greenplum、Treadata、Oracle RAC)通常会遇到两个问题:

- 更新的 throughput 不高。
- 更新影响查询。

为了解决这两个问题, Google 的 Mesa 系统设计了一个 MVCC 的数据模型, 通过增量更新和合并技术, 将离散的更新 I/O 转变成批量 I/O, 平衡了查询和更新的冲突, 提高了更新的吞吐量。

Mesa 设计了一个多版本管理技术来解决更新的问题:

- 使用二维表来管理数据, 每张表都要制定 Schema, 类似于传统的数据库。
- 每个字段用 Key/Value 来管理。Schema 就是 Key 的集合。
- 每个字段指定一个聚合函数 F (最常见的是 SUM)。

^① 参考 <http://jiezhuzhu2007.iteye.com/blog/2265152>。

- 数据更新进来的时候，按照 MVCC 增量更新，并给增量更新指定一个版本号 N 和谓词 P。
- 查询进来的时候，自动识别聚合函数，把所有版本的更新按照聚合函数自动计算出来。
- 多版本如果永远不合并，则存储的代价会非常大。而且因为每次查询需要遍历所有版本号，所以版本过多会影响查询。因此，定期合并是必需的。
- Mesa 采用两段更新的策略。更新数据按版本号实时写入，每 10 个版本自动合并；每天全量合并一遍，合并成一个基础版本。

2. 索引技术

索引是关系型数据库里的重要概念。总的来说，索引就是拿空间换时间。数据库技术和大数据技术会有一个融合的过程，除了前面讲到的 B 数索引、Hash 索引等，还有倒排索引、MinMax 索引、BitSet 索引、MDK 索引等。

大数据的核心是“大”，大数据索引和传统索引最主要的不同考虑点也是数据量的级别增大后索引本身也会变得很大。传统的 B 树索引是一个全局索引，数据量增大后，可能一台物理机的内存根本无法装下索引本身，每次插入之后，索引更新的代价会大到无法接受。索引本身的分布式需要充分考虑。

另外一个变化就是很多索引不再单独存储。有一种思路就是，数据本身以索引的形式存储下来，需要的时候才加载到内存中，而不是传统实现里将全部索引装载到内存中。

1) 倒排索引

在一个未经处理的数据库中，一般以文档 ID 作为索引，以文档内容作为记录。而 Inverted Index 指的是将单词或记录作为索引，将文档 ID 作为记录，这样便可以方便地通过单词或记录查找到其所在的文档。

2) Lucene 倒排索引原理

Lucene 是一个高性能的 Java 全文检索工具包，它使用的是倒排文件索引结构。该结构及相应的生成算法如下。

(1) 设有两篇文章 1 和 2。

文章 1 的内容为：Tom lives in Guangzhou, I live in Guangzhou too。

文章 2 的内容为：He once lived in Shanghai。

(2) 由于 Lucene 是基于关键词索引和查询的，所以首先要取得这两篇文章的关键词。通常的处理措施如下：

a. 我们现在拥有的是文章内容，即一个字符串，先要找出字符串中的所有单词，即分词。英文单词由于用空格分隔，所以比较好处理。中文单词是连在一起的，因而需要特殊的分词处理。

b. 文章中的“in”、“once”、“too”等词没有什么实际意义，中文中的“的”、“是”等字通常也无具体含义，这些不代表概念的词可以过滤掉。

c. 用户通常希望查“He”时能把含“he”、“HE”的文章也找出来，所以所有单词需要统一大小写。

d. 用户通常希望查“live”时能把含“lives”、“lived”的文章也找出来，所以需要把“lives”、“lived”

还原成 “live”。

e. 文章中的标点符号通常不表示某种概念，也可以过滤掉。

在 Lucene 中，以上措施由 Analyzer 类完成。经过处理后，文章 1 的所有关键词为：[tom][live][guangzhou][i][live][guangzhou]；文章 2 的所有关键词为：[he][live][shanghai]。

(3) 有了关键词后，我们就可以建立倒排索引了。上面的对应关系是：“文章号”对“文章中所有关键词”。倒排索引把这个关系倒过来，变成：“关键词”对“拥有该关键词的所有文章号”。文章 1、2 经过倒排后变成：

关键词	文章号
Guangzhou	1
He	2
I	1
Live	1,2
Shanghai	2
Tom	1

通常仅知道关键词在哪些文章中出现还不够，还需要知道关键词在文章中的出现频率和出现位置。通常有两种位置：

- 字符位置，即记录该词是文章中第几个字符（优点是关键词亮显时定位快）。
- 关键词位置，即记录该词是文章中第几个关键词（优点是节约索引空间、词组查询快）。Lucene 中记录的就是这种位置。

加上“出现频率”和“出现位置”信息后，索引结构变为：

关键词	文章号[出现频率]	出现位置
guangzhou	1[2]	3,6
he	2[1]	1
i	1[1]	4
live	1[2], 2[1]	2,5,2
shanghai	2[1]	3
tom	1[1]	1

以 live 这一行为例说明一下该结构：live 在文章 1 中出现了 2 次，在文章 2 中出现了 1 次，它的出现位置为“2,5,2”。这表示什么呢？我们需要结合文章号和出现频率来分析。文章 1 中出现了 2 次，那么“2,5”就表示 live 在文章 1 中出现的两个位置；在文章 2 中出现了一次，剩下的“2”就表示 live 是文章 2 中的第 2 个关键字。

以上就是 Lucene 索引结构中最核心的部分。我们注意到关键字是按字符顺序排列的（Lucene 没有使用 B 树结构），因此，Lucene 可以用二元搜索算法快速定位关键词。

实现时，Lucene 将上面三列分别作为词典文件（TermDictionary）、频率文件（Frequencies）、位置文件（Positions）保存。其中词典文件不仅保存了每个关键词，还保存了指向频率文件和位置文件

的指针,通过这些指针可以找到该关键字的频率信息和位置信息。

Lucene 中使用了 Field 的概念,用于表达信息所在位置(如标题中、文章中、URL 中)。在创建索引时,该 Field 信息也记录在词典文件中,每个关键词都有一个 Field 信息(因为每个关键字一定属于一个或多个 Field)。

为了减小索引文件的大小, Lucene 对索引使用了压缩技术。首先,对词典文件中的关键词进行压缩,关键词压缩为<前缀长度,后缀>。例如,当前词为“阿拉伯语”,上一个词为“阿拉伯”,那么“阿拉伯语”被压缩为<3,语>。其次,大量用到的是对数字的压缩,数字只保存与上一个值的差值(这样就可以减小数字的长度,进而减少保存该数字所需的字节数)。例如,当前文章号是 16389(不压缩要用 3 字节保存),上一个文章号是 16382,压缩后保存 7(只用 1 字节)。

下面通过对该索引的查询来解释一下为什么要建立索引。

假设要查询单词“live”, Lucene 先对词典进行二元查找,找到该词后,通过指向频率文件的指针读出所有文章号,然后返回结果。词典通常非常小,因而整个查询过程的时间是毫秒级的。

而用普通的顺序匹配算法,不创建索引,而是对所有文章的内容进行字符串匹配。这一过程将会相当缓慢,当文章数目很大时,所需时间往往是无法忍受的。

3) 正向索引和倒排索引的联系与区别

正向索引是经过搜索引擎对页面文本分词、消噪、去重、提取关键词后,得到的能够反映页面主体内容的一个关键词组成的集合。同时记录每一个关键词在页面上的出现频率、出现次数、格式、位置。这样,每个页面都可以被记录为一个关键词的元组,其中包含每个关键词的词频、格式、位置等权重信息。

正向索引不能直接用于排名。如果只存在正向索引,那么排名程序需要扫描所有索引库中的文件,找出包含关键词的文件,再进行相关性计算,这样的计算量无法满足实时返回排名结果的要求。

所以搜索引擎会将正向索引数据仓库重新构造为倒排索引,把文件到关键词的映射转换为关键词到文件的映射。在倒排索引中,关键词是主键,每个关键词都对应一系列文件,这些文件中都出现了这个关键词。这样,当用户搜索某个关键词时,排序程序在倒排索引中定位到这个关键词,就可以立即找出所有包含这个关键词的文件。

10.6.2 缓存技术

在计算机体系里,内存的速度要远高于磁盘,所以有了合理的数据组织。有了索引,还不够快,自然而然地想到了缓存。缓存的本质就是将核心的数据提前放到内存中,充分利用内存的高速度。缓存里面有很多关键技术,不展开来讲,只介绍一个当前比较热门的缓存组件 Tachyon。

与 HDFS 不同, Tachyon 是一个高容错的分布式文件系统,允许文件以内存的速度在集群框架中进行可靠的共享,就像 Spark 和 MapReduce 那样。通过利用信息继承、内存侵入, Tachyon 获得了高性能。Tachyon 工作集文件缓存在内存中,并且让不同的 Jobs/Queries 及框架都能以内存的速度来访

间缓存文件。因此，Tachyon 可以减少那些需要经常使用的数据集通过访问磁盘来获得的次数。

从 Tachyon 的设计目标来看，其主要提供一个基于内存的分布式的文件共享框架，不仅需要具备容错能力，还要体现内存的性能优势。

Tachyon 以常见的 Master/Worker 方式组织集群，由 Master 节点负责管理维护文件系统 MetaData，文件数据维护在 Worker 节点的内存中。

在容错性方面，主要的技术要点包括：

- 底层支持 Pluggable 的文件系统，如 HDFS 用于用户指定文件的持久化。
- 使用 Journal 机制持久化文件系统的 MetaData。
- 使用 ZooKeeper 构建 Master 的 HA。
- 没有使用 Replica 复制内存数据，而是采用和 Spark RDD 类似的 Lineage 思想用于灾难恢复。

此外，为了兼容 Hadoop 应用，提供了与 HDFS 兼容的 API 接口。

Tachyon 作为数据缓冲层，Spark+Tachyon 结合作为统一数据分析平台。Pivotal 公司用 Spark+Tachyon 作为数据湖（Data Lake）的解决方案，加快统一分析的速度。

10.7 小结

存储是一切的基础，数据要用得好、用得快，就要根据业务的特点采用不同的数据组织技术。

第 11 章

大数据云化

云计算发展到今天，可以说已经成功地从概念落地到实际。企业 IT 系统是否上云，已经成为企业 CIO 构建企业 IT 系统优先考虑的问题。以 AWS/Microsoft Azure 为代表的厂商，每年的云计算收入高达几十亿美元，而且仍能保持较快的增长速度。从广义上说，大数据技术也是云上的一种基础服务。大数据技术怎样服务化是一个值得研究的领域。本章将讨论以 AWS 为代表的 EMR 方案、阿里的 ODPS 等众多厂商探讨的服务化方案，以及 Docker 技术对大数据的影响等内容。

11.1 云计算定义

对云计算的定义有多种说法。现阶段广为接受的是美国国家标准与技术研究院 (NIST) 的定义：云计算是一种按使用量付费的模式，这种模式提供可用的、便捷的、按需的网络访问，进入可配置的计算资源共享池（资源包括网络、服务器、存储、应用软件、服务），这些资源能够被快速提供，只需投入很少的管理工作，或服务供应商进行很少的交互。从这个定义可以看出，云计算更多的是一种新的企业 IT 商业模式的转变和创新，从而带来技术的变更。

云计算包括以下几个层次的服务。

(1) IaaS：基础设施即服务。消费者通过 Internet 可以从完善的计算机基础设施获得服务。例如，硬件服务器租用。

(2) PaaS：平台即服务。PaaS 实际上是指将软件研发的平台作为一种服务，以 SaaS 的模式提交给用户。因此，PaaS 也是 SaaS 模式的一种应用。但是，PaaS 的出现可以加快 SaaS 的发展，尤其是加快 SaaS 应用的开发速度。例如，软件的个性化定制开发。

(3) SaaS：软件即服务。它是一种通过 Internet 提供软件的模式，用户无须购买软件，而是向服务提供商租用基于 Web 的软件来管理企业的经营活动。

11.2 应用上云

在介绍大数据云化架构之前，先来讨论一下云化的基本知识和基本概念。

11.2.1 Cloud Native 概念

云计算流行开来，读者经常会听到 Cloud Native，那什么才谈得上 Cloud Native 呢？

从概念上讲，核心说的是传统的应用部署在数据中心上的架构不适合云化的环境，既要充分利用云基础设施的可编程性和扩展性，又要规避云基础设施的不可靠，Cloud Native 的核心改变是应用适合基础设施，而不是基础设施适合应用。

什么样的应用才算是 Cloud Native 呢？Open Data Center Alliance (ODCA) 的一篇论文 *Best Practices: Architecting Cloud-Aware Applications Rev. 1.0* 给出了一个成熟度模型，如图 11.1 所示。

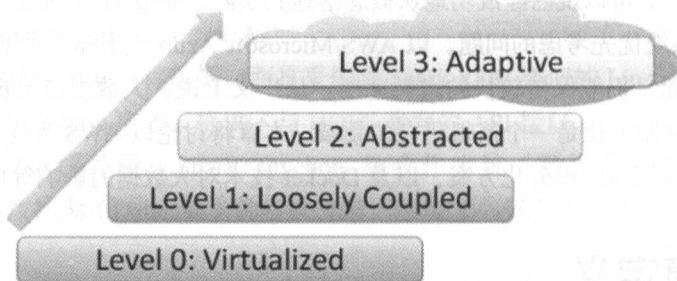


图 11.1

(1) Level 0: Virtualized。0 级是最低要求，应用可以快速和很容易地部署到云上的虚拟机。

(2) Level 1: Loosely Coupled。主要的应用都是松耦合的，一个比较好的步骤是：

第一步，和数据存储松耦合，包括配置数据、日志等。

第二步，和网络松耦合，使用 NamingService 而不是 IP 地址和端口号。

(3) Level 2: Abstracted。这一级别，服务与架构完全是松耦合的。微服务架构是一个比较好的应用例子。

(4) Level 3: Adaptive。这一级别，应用必须有足够的自动化能力。

11.2.2 微服务架构^①

1. 软件开发过程中遇到的问题

一个简单的应用会随着时间的推移逐渐变大。在每次的敏捷开发过程中，开发团队都会面对新“故事”，然后开发许多新代码。几年后，这个小而简单的应用会变成一个巨大的“怪物”。

一旦你的应用变成一个又大又复杂的“怪物”，那开发团队肯定很痛苦。敏捷开发和部署举步维艰，其中最主要的问题就是这个应用太复杂，以至于任何单个开发者都不可能搞懂它。因此，修正 bug 和正确地添加新功能变得非常困难，并且非常耗时。如果代码难以理解，就不可能被正确地修改，最终会走向巨大的、不可理解的泥潭。

单体式应用也会降低开发速度。应用越大，启动时间越长。调查表明，有时候应用的启动时间

^① 参考 <http://www.d1net.com/cloud/news/351811.html> 和 <http://www.infoq.com/cn/news/2013/12/micro-service-architecture/>。

竟然超过 12 分钟，某些应用甚至需要 40 分钟的启动时间。如果开发者需要经常重启应用，那么大部分时间就要在等待中度过，生产效率会受到极大影响。

另外，复杂而巨大的单体式应用也不利于持续性开发。目前，SaaS 的应用常态就是每天改变很多次，而这对于单体式应用模式非常困难。另外，这种变化带来的影响并没有很好地被理解，所以不得不做很多手工测试。那么，接下来，持续部署也会很艰难。

单体式应用在不同模块发生资源冲突时，扩展将会非常困难。比如，一个模块完成一个 CPU 敏感逻辑，应该部署在 AWS EC2 计算优化型实例，而另外一个内存数据库模块更适合 EC2 内存优化型实例。然而，由于这些模块都部署在一起，因此，不得不在硬件选择上做出妥协。

单体式应用的另外一个问题是可靠性。因为所有模块都运行在一个进程中，任何一个模块中的一个 bug，如内存泄漏，都有可能拖垮整个进程。除此之外，因为所有的应用实例都是唯一的，这个 bug 将会影响到整个应用的可靠性。

最后，单体式应用使得采用新架构和语言非常困难。比如，设想有两百万行采用 XYZ 框架编写的代码，如果想改成 ABC 框架，无论是时间还是成本都是非常昂贵的。因此，这是一个无法逾越的鸿沟。

总结一下：一开始有一个很成功的关键业务应用，后来变成一个巨大的、无法理解的“怪物”。因为采用过时的、效率低的技术，使得雇佣有潜力的开发者很困难，导致应用无法扩展、可靠性很低，最终，敏捷性开发和部署变得无法完成。

2. 微服务解决应用复杂性

通过采用微服务解决了上述问题。其思路不是开发一个巨大的单体式的应用，而是将应用分解为小的、互相连接的微服务。

一个微服务一般完成某个特定的功能，如下单管理、客户管理等。每个微服务都是微型六角形应用，都有自己的业务逻辑和适配器。一些微服务还会发布 API 给其他微服务和应用客户端使用。其他微服务完成一个 Web UI，运行时，每个实例可能是一个云 VM 或者 Docker 容器。

运行时，行程管理服务由多个服务实例构成，每个服务实例都是一个 Docker 容器。为了保证高可用，这些容器一般都运行在多个云 VM 上。在服务实例前部署 Nginx 负载均衡器，它们负责在各个实例间分发请求。负载均衡器也同时处理其他请求，如缓存、权限控制、API 统计和监控等。

这种微服务架构模式深刻地影响了应用和数据库之间的关系，不像传统的多个服务共享一个数据库，微服务架构的每个服务都有自己的数据库。另外，这种思路也影响到了企业级数据模式。同时，这种模式意味着多份数据。但是，如果想获得微服务带来的好处，那么每个服务独有一个数据库是必需的，因为这种架构需要这种松耦合。

3. 到底什么是微服务

实际上微服务本身并没有一个严格的定义，普遍的观点是：微服务是一种简单的应用，有 10~100 行代码。有一点需要注意，那就是微服务通常都是很小的，甚至是微型的。这意味着你不会在大

型框架上看到很多微服务，因为这是不切实际的。简单与轻量级是当今的主流。

从物理角度来说，这些服务都很小，可以在同一台机器上运行大量服务，而不必担心内存或资源等问题。重申一遍，基于大型框架的简单库将会取得最后的胜利，而对第三方库的依赖越来越小。

这种服务层上的解耦还提供了另外一种有趣的选择。我们将很多老旧应用的复杂性推到了基础设施层，不再受限于单个技术栈或语言。我们现在可以发挥出任何技术栈或语言的优势。

你不会看到任何基于微服务的架构是托管在应用服务器上的，这是关键。本质上，微服务是自我托管的，它们获取一个端口，然后监听。这意味着你将失去典型的企业应用服务器所带来的很多好处，服务器需要提供这些必要的功能（如性能度量、监控等）。

4. 微服务的通信

服务之间该如何通信呢？这个问题是无法通过一个简单的答案解决的，甚至在单个解决方案中也是难以做到的。最基本的答案就是通过 HTTP 公开所有服务，然后以 JSON 作为数据交换格式。服务探测（一个服务是如何找到另一个服务的）可以很简单，只需将端点细节信息放到配置文件中即可（硬编码也可以）。

你可能会发现，在某些情况下，一个完整事务中对 JSON 负载的序列化与反序列化的代价会造成系统瓶颈。也许 JSON 并不适合，你可能需要使用其他协议，如 Protobuf 等。

不过硬编码的 URL 会导致耦合。在分层应用中，这是有意义的，不过对于基于服务的架构来说，这意味着你不能再被这种潜在的限制所约束。服务之间的某些通信可以是完全解耦的，事实上，有些服务可以随意发布事件或数据。它们只是将其扔出去（比如说消息总线），也许有一天出现了某个服务，然后开始监听了。此外，也许系统的某些部分会以批量/离线的流程运作，它们可能会在几小时后才从队列中取出消息。微服务架构可以实现这种灵活性而无需修改整个架构。

5. 微服务的部署

部署一个微服务应用也很复杂，一个分布式应用只需简单地在复杂均衡器后面部署各自的服务器即可。每个应用实例需要配置诸如数据库和消息中间件等基础服务。相对而言，一个微服务应用一般由大批服务构成。每个服务都有多个实例，这就造成许多需要配置、部署、扩展和监控的部分。除此之外，还需要完成一个服务发现机制，用来发现与它通信服务的地址（包括服务器地址和端口）。传统的解决问题的办法不能用于解决这么复杂的问题。接踵而来，成功部署一个微服务应用需要开发者有足够的控制部署方法，并高度自动化。

一种自动化方法是使用 PaaS 服务，如 CloudFoundry。PaaS 给开发者提供了一个部署和管理微服务的简单方法，它把所有问题都打包内置解决了。同时，配置 PaaS 的系统和网络专家可以采用最佳实践和策略来简化问题。另外，一个自动部署微服务应用的方法是开发最基础的 PaaS 系统。一个典型的开始点是使用一个集群化方案，比如配合 Docker 使用 Mesos 或者 Kubernetes。

6. 监控与度量

分层解决方案的组件出现问题时不会销声匿迹，要么是编译失败，要么是遇到问题时抛出异常

(除非将抛出的异常隐藏了)。在基于服务的方式中,有的服务可能出现了问题,而其他服务则很容易发现问题(特别是在 pub/sub 模型中)。这意味着我们必须能对服务进行监控和编排。事实上,只知道服务还能用是不够的,还要知道服务是不是还能提供业务价值、能否继续使用、是否是可靠交易的瓶颈。

监控是非常重要的事情,对于基于服务的架构来说更是如此,因为这时出现的失败并不容易被发现。JVM 世界中的 Metrics 与 Ostrich 等库不仅可以收集度量信息,还提供了与 Nagios 和 Ganglia 等服务的集成,可以将数据发送给它们。

7. 微服务的测试

对于基于微服务架构的系统来说,测试服务并没有什么特殊之处,不过这里要强调的是不必再对每个服务使用完整的测试套件。因为一个服务只做一件事,因此引入系统 Bug 的概率明显降低,这要归功于基于服务的系统的天生行为。

8. 微服务架构的好处

微服务架构模式有很多好处。首先,通过分解巨大单体式应用,为多个服务方法解决了复杂性问题。在功能不变的情况下,应用被分解为多个可管理的分支或服务。每个服务都有一个用 RPC 或者消息驱动 API 定义清楚的边界。微服务架构模式给采用单体式编码方式很难实现的功能提供了模块化的解决方案,由此,单个服务很容易被开发、理解和维护。

其次,这种架构使得每个服务都可以有专门的开发团队来开发。开发者可以自由选择开发技术,提供 API 服务。当然,许多公司试图避免混乱,只提供某些技术选择。这种自由意味着开发者不需要被迫使用某项目开始时采用的过时技术,他们可以选择现在的技术。甚至于,因为服务都是相对简单的,即使用现在的技术重写以前的代码也不是很困难的事情。

再次,微服务架构模式是每个微服务独立的部署。开发者不再需要协调其他服务部署对本服务的影响。这种改变可以加快部署速度。UI 团队可以采用 AB 测试,快速地部署变化。微服务架构模式使得持续化部署成为可能。

最后,微服务架构模式使得每个服务独立扩展。可以根据每个服务的规模来部署满足需求的规模,可以使用更适合服务资源需求的硬件。

9. 微服务架构的不足

像任何其他科技一样,微服务架构也有不足。首先,微服务强调了服务大小。尽管小服务更乐于被采用,但是不要忘了这只是终端的选择,而不是最终的目的。微服务的目的是有效地拆分应用,实现敏捷开发和部署。

其次,微服务应用是分布式系统,由此会带来固有的复杂性。开发者需要在 RPC 或者消息传递之间选择并完成进程间通信机制。更甚于,他们必须写代码来处理消息传递中速度过慢或者不可用等局部失效问题。

一个关于微服务的挑战来自分区的数据库架构。商业交易中同时给多个业务分主体更新消息很

普遍。这种交易对于单体式应用来说很容易，因为只有一个数据库。在微服务架构应用中，需要更新不同服务所使用的不同的数据库。使用分布式交易并不一定是好的选择，不仅仅是因为 CAP 理论，还因为今天高扩展性的 NoSQL 数据库和消息传递中间件并不支持这一需求，最终不得不使用一个一致性的方法，从而对开发者提出了更高的要求和挑战。

再次，测试一个基于微服务架构的应用也是很复杂的任务。比如采用流行的 SpringBoot 架构，对一个单体式 Web 应用，测试它的 REST API 是很容易的事情。反过来，同样的服务测试需要启动和它有关的所有服务。再重申一次，不能低估了采用微服务架构带来的复杂性。

最后，微服务架构模式应用的改变将会波及多个服务。假设你在完成一个案例，需要修改服务 A、B、C，而 A 依赖 B，B 依赖 C。在单体式应用中，只需改变相关模块、整合变化、部署即可。对比之下，微服务架构模式就需要考虑相关改变对不同服务的影响。比如，你需要先更新服务 C，然后是 B，最后才是 A。幸运的是，许多改变一般只影响一个服务，而需要协调多服务的改变很少。

10. 微服务总结

(1) 微服务是一种新的架构思路，从传统的 SOA 中发扬出来，摒弃了传统 SOA 重型、低效的弊端。

(2) 微服务更多的是一种架构理念，服务的划分没有一个强规则，遵循服务自治、数据自有即可。实施微服务的同时也在践行新的敏捷、DevOps 等开发理念。

(3) 微服务适合复杂应用，就是为解耦复杂应用而诞生的。“复杂”是业务逻辑复杂，并不是资源消耗多。所以微服务更适合应用层业务的实施，而不是底层平台，如存储、数据库、OS 等。

(4) 微服务需要发展和配合更多的分布式监控、部署、测试、定位工具，架构的解耦同时引入了分布式的复杂性。

11.2.3 Docker 配合微服务架构^①

1. Docker 简介

微服务架构中服务的划分完全依照业务自身，越轻量越好，很多服务就是一个进程。微服务架构的成功应用依赖 Docker 等技术对资源进行隔离和快速部署。Docker 和微服务并不是一个紧耦合的关系，微服务不一定要部署在 Docker 上，只是说 Docker 非常适合微服务。

接下来介绍一下到底什么是 Docker。

简单地讲，Docker 是一个构建在 LXC 之上的、基于进程容器（Process Container）的轻量级 VM 解决方案。以现实世界中货物的运输作类比，为了解决各种型号、规格、尺寸的货物在各种运输工具上进行运输的问题，我们发明了集装箱。

Docker 的初衷也就是将各种应用程序和它们所依赖的运行环境打包成标准的 Container/Image，进而发布到不同的平台上运行，如图 11.2 所示。

^① 参见 <http://blog.csdn.net/zhangjun2915>。

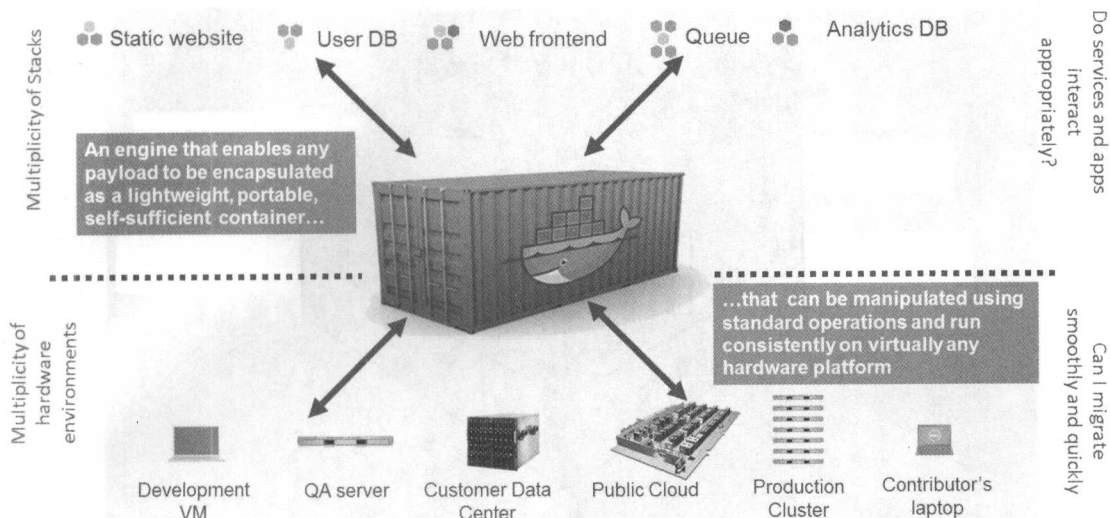


图 11.2

从理论上说，这一概念并不新鲜，各种虚拟机 Image 也起着类似的作用。Docker Container 和普通虚拟机 Image 相比，最大的区别是它并不包含操作系统内核，如图 11.3 所示。

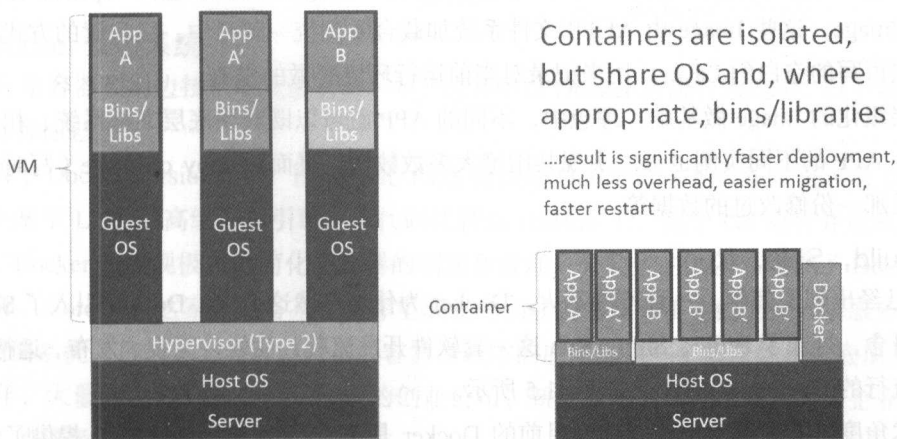


图 11.3

普通虚拟机将整个操作系统运行在虚拟的硬件平台上，进而提供完整的运行环境供应用程序运行；而 Docker 则直接在宿主平台上加载运行应用程序。本质上它在底层使用 LXC 启动一个 Linux Container，通过 CGroup 等机制对不同的 Container 内运行的应用程序进行隔离、权限管理和 Quota 分配等。每个 Container 拥有自己独立的各种命名空间（资源），包括 PID 进程、MNT 文件系统、NET 网络、IPC、UTS 主机名等。

2. Container 构建方案

除了 LXC，Docker 的核心思想就体现在它的运行容器构建方案上，如图 11.4 所示。

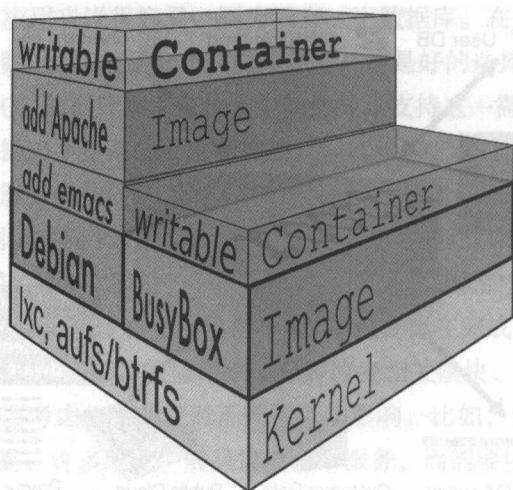


图 11.4

为了最大化地重用 Image，加快运行速度，减少内存和磁盘封装，Docker Container 运行时所构造的运行环境实际上是由具有依赖关系的多个 Layer 组成的。例如，一个 Apache 的运行环境可能是在 rootfsimage 的基础上叠加包含 Emacs 等各种工具的 Image，再叠加包含 Apache 及其相关依赖 Library 的 Image，这些 Image 由 AUFS 文件系统加载合并到统一路径中，以只读的方式存在，最后再叠加一层可写的空白的 Layer，用来记录对当前运行环境所做的修改。

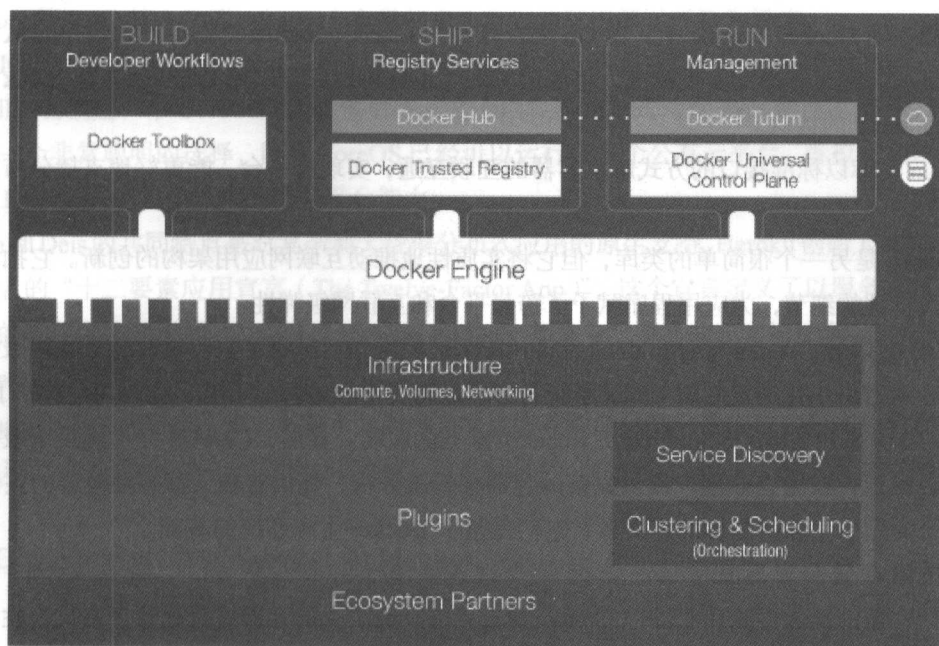
有了层级化的 Image 做基础，理想中，不同的 APP 就可以既共用底层文件系统、相关依赖工具等，同一个 APP 的不同实例也可以实现共用绝大多数数据，进而以 copy on write（写时复制）的形式维护自己那一份修改过的数据等。

3. Build, Ship, Run^①

LXC 已经出现了多年，一直不温不火，Docker 为什么突然这么火？Docker 引入了 Ship（发布）这个重要概念，打通了 Build、Ship、Run 这一套软件开发流程，使软件开发、发布、运行变得简单，契合当前流行的 DevOps 的理念，如图 11.5 所示。

从技术角度来看，基本上可以认为目前的 Docker 是 LXC 的一个高级封装，提供了各种辅助工具和标准接口，可以依靠 LXC 和各种脚本实现与 Docker 类似的功能。在实际使用中，一般不用关心底层 LXC 的细节，同时也不排除将来 Docker 实现基于非 LXC 方案的可能性。在 LXC 的基础上，Docker 额外提供的 Feature 包括标准统一的打包部署运行方案、历史版本控制、Image 重用、Image 共享发布等。

^① 参见 <http://jiezhuzhu2007.iteye.com/blog/2235661>。

图 11.5^①

4. Docker 生态系统

2013 年是容器和周边技术高歌猛进的一年，这其中以 Docker 的流行为代表。以下两家公司和他们的产品具有标志意义。

2013 年，Docker Version 0.10: Docker 是 PaaS 提供商 DotCloud（现已正式更名为 Docker Inc.）开源的一个基于 LXC 的高级容器引擎，源代码托管在 GitHub 上，基于 Go 语言并遵从 Apache 2.0 协议开源。Docker 的出现极大地简化了容器的创建和管理，分层式的 AUFS 实现了 Docker 镜像。

2013 年，CoreOS: 这家在硅谷某个车库里成立的创业公司发布了专门为大规模服务器部署定制的 Linux 精简系统，目的是为运行以轻量级容器为载体的应用提供一个高度优化的底层系统。

2014 年，大量围绕 Docker 和 CoreOS 的创业公司、新近开源的软件项目、大型企业和互联网公司的加入，使得轻量级容器技术更上一层楼。

下面从几个不同的角度来看 Docker 的生态系统。

1) Docker 和容器宿主

前文提到的 Docker Inc. 和 CoreOS 已经赚足眼球，投资者接踵而至，大规模融资此起彼伏。企业级厂商如 RedHat、Ubuntu 等不甘寂寞，纷纷亮明旗帜，选择站队。

在旧金山举行的 DockerCon 2014 展示了 Docker 对未来的雄心壮志。在 Docker Engine 逐渐稳定并标准化的背景下，Docker 的未来目标是为互联网基础架构制定新的标准。三个开源项目 Libcontainer、Libchan 和 Libswarm，吹响了这场战役的冲锋号。

^① 图片来源：<http://www.docker.com/>。

在新版本的 Docker Engine 中，由 Go 语言开发的 Libcontainer 库已取代 LXC。笔者认为，它更大的目的是反向定义容器的实现标准，将底层实现（也许可以完全不用 CGroup 甚至 Linux）都抽象化到 Libcontainer 的接口。

Libchan 类库以标准接口的方式解决容器的互联互通，实现跨平台，能更好地支持分布式系统和并发编程。

Libswarm 是另一个很简单的类库，但它将实质性地推动互联网应用架构的创新。它抽象了应用部署和集群管理的细节，为应用程序赋予了跨云平台和互联网级弹性。

CoreOS 的口号“A new way to think about servers”，这句话阐明了它们对改造互联网服务器的目标。CoreOS 通过最小化的定制版 Linux 系统为容器运行提供载体。2014 年 8 月 14 日，传来了 CoreOS 收购 Quay.IO 并推出 CoreOS Enterprise Registry 服务的新闻。显然，CoreOS 并不满足于服务器层的工作，其目标定位在为企业用户提供完整的容器技术服务栈，提供管理大型容器集群的整体解决方案。在这个类别中生存的是标准定义者，它们是整个 Docker 生态系统的基础。

2) 镜像存储和容器托管

这包括容器的镜像存储和 CaaS（Container as a Service）类的容器运行托管，有代表性的公司是 StackDock、Orchard、Tutum、Quay.IO、Baremetal.IO 等。

这几家几乎都是创业公司，它们围绕轻量级容器的整个生命周期来设计自己的产品，有的聚焦容器镜像描述文件（Dockerfile）向导化生成和构建过程的优化（如 StackDock），有的提供包括 SSD 在内的高性能托管环境（如 StackDock 和 Tutum），有的在监控和弹性扩展方面做足文章（如 Tutum），也有像 Baremetal.IO 这样针对企业级整体解决方案的公司。

容器的镜像存储和运行托管是 Docker 生态体系中非常接近最终用户的一层。这个类别中的公司也许并没有高深莫测的技术，也不是标准的定义者，但通过它们与细分市场中客户的长期沟通合作，积累了大量 Docker 商用化的经验和实践。

3) 基于 Docker 的微 PaaS

镜像存储（静态）和容器托管（动态）都是以容器为单位的。下面讲述以应用为单位、以容器为底层技术实现的微 PaaS。

随着 Microsoft Azure、Cloud Foundry 的普及，PaaS 的概念已经深入人心。在传统意义上，PaaS 实例一般都与一个特定的 IaaS 平台绑定，提供部署接口、负载平衡、服务绑定等。然而 Docker 世界中产生的微 PaaS，在此基础上进一步创新。这个领域比较有代表性的是 Flynn 和 Deis.IO，它们都是开源项目。

Flynn 分为 Layer 0 和 Layer 1 两层。Layer 0 主要做底层硬件和云平台的抽象、分布式配置、任务调度、服务发现等基础工作，它为上层的容器运行环境提供了一个抽象的资源平台。Flynn 可以快速部署在 AWS 上，今后也可以扩展到其他公有云和私有云。Layer 1 主要服务于应用，是 PaaS 功能的具体实现层，它提供了基本的管理 API 和客户端，实现了 Git Receiver、Heroku Buildpacks、Routing

和 DataStore 等 PaaS 核心功能。Layer 1 本身和它所管理的应用都以容器为载体。

Deis.IO 的一个亮点是用 CoreOS 承担底层资源管理的任务。在部署 Deis PaaS 环境时,首先安装的 Controller 会创建一个 CoreOS 系统,然后在其之上以容器的方式运行 Deis 的所有组件。对 CoreOS 的支持是一个非常聪明的选择,目前 CoreOS 已经可以运行在多个公有云平台、虚拟机和物理机环境下,这为 Deis 提供了与生俱来的跨云平台能力。

Flynn 和 Deis 的共同特点是对复杂和大规模分布式应用的原生支持。Heroku 创始人 Adam Wiggins 曾发布著名的“十二要素应用宣言(The Twelve-Factor App)”,这个宣言定义了以服务方式和通过互联网交付的软件应该遵循的 12 个要素。Flynn 和 Deis 都是十二要素的忠实拥护者,它们的微 PaaS 平台与 Heroku 有极好的兼容性。

微 PaaS 创业公司层出不穷,竞争十分激烈,但也许走到最后的只是少数。在这一轮容器技术热潮中,微 PaaS 正在影响软件开发和运维流程,改变软件的交付方式,把十二要素类互联网应用架构标准化。

4) Orchestration、Management 和 Monitor

围绕 Docker API 做 Web UI 的门槛相对较低,受到了大家的追捧。这一类主要有 DockerUI、Shipyards、maDocker 等,它们无一例外地都调用 Docker API 和其他类库,把对容器的管理和监控呈现在 Web 页面中,这在某种程度上降低了企业网管对这些新技术的恐惧。

这一领域有三个不得不提的“高富帅”项目:Google Kubernetes、CloudFoundry 的 BOSH 和 Diego。

Kubernetes 是构建在 Docker 之上的容器集群管理系统,Google 于 2014 年 6 月将这个项目开源。它可以为用户提供跨平台的处理能力,不但能够在 Google 的基础架构中运行,同时可以访问其他的云计算服务器,如 AWS,甚至是私有云。

这个系统一经开源,就得到了 IBM、RedHat、微软、Docker、Mesosphere、CoreOS 和 SaltStack 等厂商的支持:微软将确保 Kubernetes 能够在其 Azure 云中作为基于 Linux 的虚拟机系统容器并正常运行;RedHat 则将其引入了自己的云产品;IBM 的计划是为 Kubernetes 与 Docker 贡献代码;CoreOS 将在其操作系统发行版中为 Kubernetes 提供支持;SaltStack 正努力简化 Kubernetes 运行在其他环境下的部署流程;而 Mesosphere 则打算将这项技术加入自己的 Mesos 同名开源项目当中。Google 一呼百应的大将之风展露无遗。

CloudFoundry 的 BOSH 是部署和运维工具,它通过类似操作系统驱动程序 CPI(Cloud Provider Interface)来实现对多种异构云平台的支持和抽象,以近乎优雅的方式管理 VM 模板、软件发布(Release)和部署配置脚本文件。BOSH 还推出了一个试验性质的项目 BOSH Release for Docker。

CloudFoundry 在它的 DEA(Droplet Execution Agent)中使用基于 Warden 的容器技术来做 PaaS 的应用隔离。最新的 Diego(Go 语言版 DEA)项目目标是让 CloudFoundry 在跨运行时环境方面更具有扩展性,这些运行时环境就包括 Docker,也可能会原生支持 Windows Server。

5) 网络层的增强和解决方案

容器之间如何互联互通？Docker 引擎中的内联网络能否满足企业级别的网络需求？当容器像今天的虚拟机一样在企业环境大规模部署时，复杂的网络需求如网络配置管理、安全监控、流量 QoS、网络隔离等一定会出现。

在虚拟化的世界里，这些需求催生了庞大的网络虚拟化（SDN）产业，在容器的环境中，是否有同样的挑战和机会？在这个领域，目前受关注较多的是 SkyDock 和 VNS3 开源项目，但整体上还处于萌芽阶段。

5. 什么阻碍 Docker 的广泛应用

Docker 的理念非常好，当前技术也非常热门，但在实际产品中，真正用到 Docker 的非常少。

首先是安全问题。虚拟机的安全是经过验证的，这种轻量级的通过内存空间的隔离宿主 OS 很容易被攻破，所以目前 Docker 在公有云上的应用非常少。当然，Docker 已经在这方面进行了很多改进，比如运行时的系统调用黑名单，围绕镜像的安全也引起了关注。

其次，大型应用/复杂应用使用 Dockerfile 基本是不可能的事情，由于它抽象层次太高，以至于不能应对复杂的用例。

最后是它依赖大量的内核的新特性，而这些内核特性还远未成熟，在实际使用过程中常常出现稳定性问题。

虽说这项技术还未成熟，但总的来说，Docker 是一项非常有前景的技术。

11.2.4 应用上云小结

总的来说，云化趋势不可避免。Cloud Native 架构要求和原有单体软件的架构并不完全相同，应用上云，要根据实际情况，选择是否采用微服务架构。

11.3 大数据上云

11.3.1 大数据云服务的两种模式

大数据在云上是一种 PaaS 应用。提供大数据云服务的公司有很多，国外的有 AWS/Azure，国内的有阿里云、华为企业云、青云等。大数据和云结合，一般来说有两种典型模式。

第一种可以称之为集群模式，以 AWSEMR 为典型代表。这种模式的核心是通过云的能力简化了集群的创建、运维等。第二种可以称之为服务模式，它进一步简化了大数据的使用，用户不用关心集群、资源这些事情，只需将大数据任务提交给大数据云，享受相应的服务即可。这种模式最典型的是微软的 Azure Data Lake Analytics。先分别来看一下两种模式的架构的关键点和优缺点。

11.3.2 集群模式 AWSEMR^①

1. 架构简介

Amazon Elastic Map Reduce (Amazon EMR) 是一种 Web 服务, 让用户能够轻松、快速并经济地处理大量的数据。

Amazon EMR 简化了大数据处理, 提供的托管 Hadoop 框架可以跨越各个动态可扩展的 Amazon EC2 实例分发和处理海量数据, 如图 11.6 所示。

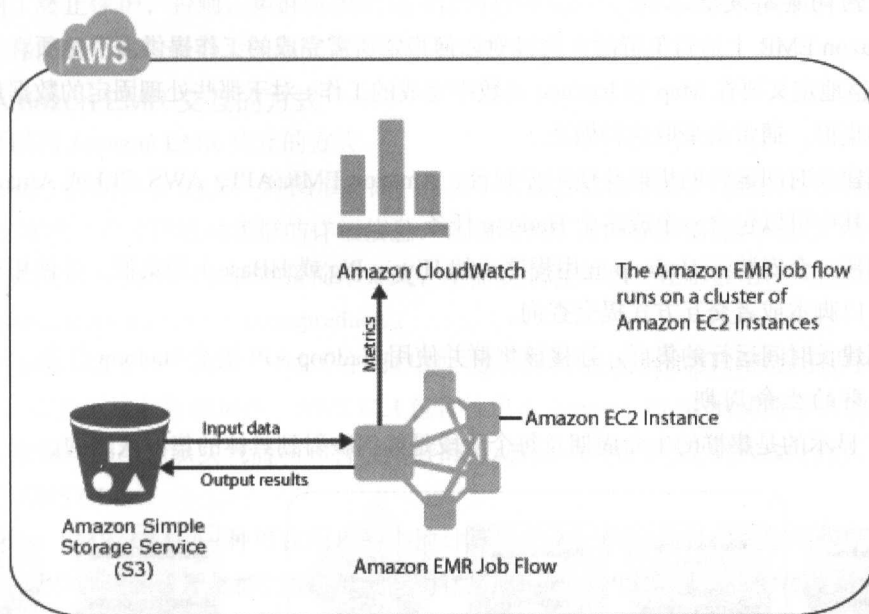


图 11.6

在 Amazon EMR 上运行的 Hadoop 集群使用 EC2 实例作为虚拟 Linux 服务器用于主节点和从属节点, 将 Amazon S3 用于输入和输出数据的批量存储, 并将 Cloud Watch 用于监控集群性能和发出警报。

AWS EMR 采用存储和计算分离的架构, 数据存储在 S3 上, 计算资源来自 EC2 实例。集群创建之后, MapReduce 通过 HDFS 代理调用 S3 接口, 从 S3 上读取和写入数据。

2. 节点

Amazon EMR 为集群中的服务器定义了三种角色。

(1) 主节点——管理集群: 协调将 MapReduce 可执行文件和原始数据子集分配到核心实例组和任务实例组。此外, 它还会跟踪每个任务的执行状态, 监控实例组的运行状况。一个集群中只有一个主节点。这与 Hadoop 主节点映射。

^① 参考 http://docs.aws.amazon.com/zh_cn/ElasticMapReduce/latest/DeveloperGuide/emr-what-is-emr.html。

(2) 核心节点——使用 Hadoop 分布式文件系统 (HDFS) 运行任务和存储数据。这与 Hadoop 从属节点映射。

(3) 任务节点 (可选)——运行任务：这与 Hadoop 从属节点映射。

3. 集群

集群是一组执行工作的服务器。在 Amazon EMR 中, 集群是一组以 EC2 实例形式运行的虚拟服务器。

1) 如何向集群发送工作

在 Amazon EMR 上运行集群时, 会针对如何指定所需完成的工作提供多个选项。

(1) 完整地定义要在 Map 和 Reduce 函数中完成的工作。对于那些处理固定的数据量并在处理完成时终止的集群, 通常会采取这种做法。

(2) 创建长时间运行的集群并使用控制台、Amazon EMR API、AWS CLI 或 Amazon EMR CLI 提交步骤, 其中可以包含一个或多个 Hadoop 任务。

(3) 创建一个安装了 Hadoop 应用程序 (如 Hive、Pig 或 HBase) 的集群, 并使用这些应用程序提供的接口以脚本或者交互方式提交查询。

(4) 创建长时间运行的集群、连接该集群并使用 Hadoop API 提交 Hadoop 任务。

2) 集群的生命周期

图 11.7 显示的是集群的生命周期及每个阶段是如何映射到具体的集群状态的。

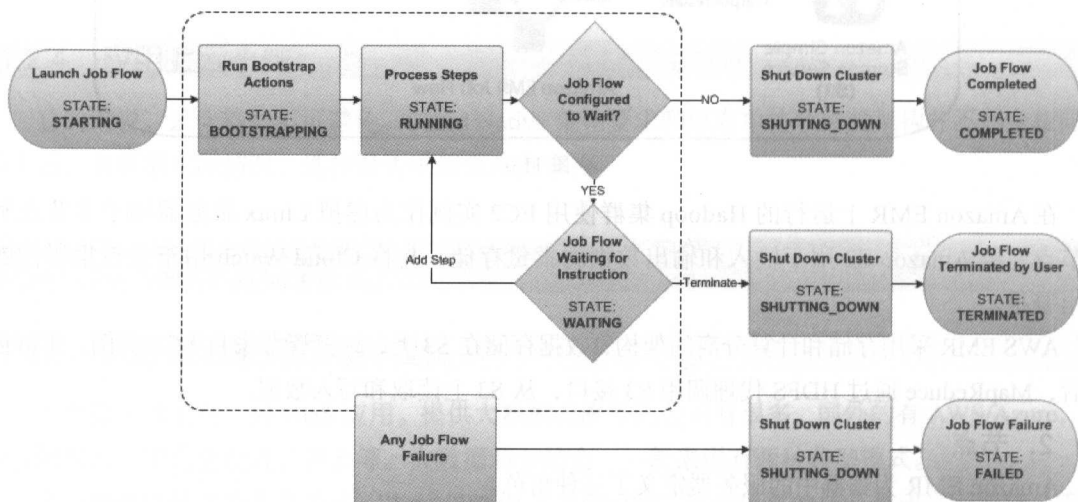


图 11.7

成功的 Amazon EMR 集群遵循此流程: Amazon EMR 先配置 Hadoop 集群, 在这期间, 集群的状态是 STARTING; 接着, 运行任何用户定义的引导操作, 在这期间, 集群的状态是 BOOTSTRAPPING; 在所有引导操作完成后, 集群的状态是 RUNNING, 在此阶段, 任务流程会按顺序运行所有的集群步骤。

如果用户通过启用 `keepalive` 参数将集群配置为长时间运行的集群,那么集群会在处理完成后等待下一组说明时进入 `WAITING` 状态。用户必须在不再需要该集群时手动终止该集群。

如果用户将集群配置为暂时性的集群,那么它将在所有的步骤完成后自动关闭。

当集群在没有遇到错误的情况下终止时,它的状态会转换为 `SHUTTING_DOWN`,且集群会关闭,从而终止虚拟服务器实例。集群上存储的所有数据都会被删除,而其他地方(如 Amazon S3 存储段)中存储的信息会保存下来。最后,当所有的集群活动完成时,集群的状态会标记为 `COMPLETED`。

除非启用了终止保护,否则,集群流程期间的任何故障都会终止该集群及其所有的虚拟服务器实例,集群上存储的任何数据都会被删除,集群的状态会标记为 `FAILED`。

4. 和 Amazon EMR 交互的方式

有多种可以和 Amazon EMR 交互的方式。

(1) Console (控制台): 这是一种图形界面,可用于启动和管理集群。借助这个界面,用户可以填写各种 Web 窗体,指定待启动集群的详细信息,查看现有集群的详细信息,调试和终止集群。使用控制台是开始使用 Amazon EMR 的最简单方式,不需要编程知识。控制台是在线提供的,网址是 <https://console.aws.amazon.com/elasticmapreduce/>。

(2) AWS CLI (命令行界面): 一种可在用户的本地计算机上运行的客户端应用程序,用于连接 Amazon EMR,以及创建和管理集群。AWS CLI 包含特定于 Amazon EMR 的功能丰富的命令集。用户可以使用它来编写脚本,以实现启动和管理集群的自动化。如果用户希望从命令行工作,则最好的选择是使用 AWS CLI。

(3) Amazon EMR CLI: 一种可在用户的本地计算机上运行的旧式客户端应用程序,用于连接 Amazon EMR,以及创建和管理集群。用户可以使用它来编写脚本,以实现启动和管理集群的自动化。Amazon EMR CLI 的功能开发已停止。我们鼓励使用 Amazon EMR CLI 的客户迁移至 AWS CLI。新用户应该下载 AWS CLI,而不是 Amazon EMR CLI。

(4) Software Development Kit (软件开发工具包, SDK): AWS 提供一个带有各种函数的软件开发工具包,这些函数会调用 Amazon EMR 创建和管理集群。借助该软件开发工具包,用户可以编写应用程序,用于自动处理集群的创建和管理流程。如果用户希望扩展或者自定义 Amazon EMR 的功能,那么软件开发工具包是最好的选择。用户可以从 <http://aws.amazon.com/sdkforjava/> 下载适用于 Java 的 AWS 开发工具包。

(5) Web Service API: AWS 提供低级别的界面,可以用来直接使用 JSON 调用 Web 服务。如果想要创建调用 Amazon EMR 的自定义软件开发工具包,则最好的选择是使用该 API。有关详细信息请参阅 Amazon EMR API Reference。

5. EMR 架构的优缺点

存储和计算分离架构,其最大的好处是集群按需创建,需要时创建集群,不需要时可以释放,从而节省成本。该架构非常适合云上按需获取资源的模式。这种方案的比较大的问题是性能比较低

下；集群创建周期长，通常需要十分钟甚至以上；虚拟机性能下降比较明显，大数据是重负载任务，通常虚拟机上的性能相比物理机集群下降 60%；集群规模受限，不能像物理机集群那样一个集群包含几千台虚拟机。这就意味着该架构只适合中小客户，只能处理几百 GB 规模的小数据。

11.3.3 服务模式 Azure Data Lake Analytics

集群托管模式解决的是大数据组件安装、运维这些事情，极大地降低了大数据使用和运维难度。但是对于客户来说，还需要理解大数据组件知识，才能规划好集群大小，写好代码。能不能更进一步呢？将资源规划、扩容、缩容这些事情都节省掉？这就是大数据服务模式所做的事情。服务模式提供多种接口能力、API、交互界面、程序，客户只需根据需求提交代码或者直接使用界面定制，运行过程中无须参与，等待最终结果即可。

服务模式相比集群托管模式在易用性上更进一步，内部核心的能力主要涉及多租户、资源弹性伸缩、统一接口等。

1. Server-less 概念

Server-less 一词来源于 AWS 的 Lambda 服务 (<https://aws.amazon.com/cn/lambda/>)，通过 AWS Lambda，无须配置或管理服务器即可运行代码。用户只需按消耗的计算时间付费，代码未运行时不产生费用。借助 Lambda，用户几乎可以为任何类型的应用程序或后端服务运行代码，而且全部无须管理。只需上传代码，Lambda 就会处理运行和扩展高可用性代码所需的一切工作。用户可以将代码设置为自动从其他 AWS 服务触发，或者直接从任何 Web 或移动应用程序调用。

其核心是客户不用关心服务器，只需管理自己的代码即可。大数据领域也能提供类似理念的服务，Azure Data Lake Analytics 就是一个例子。

2. 架构关键能力

这种模式最典型的是 Azure 的 Data Lake。可以访问 <https://azure.microsoft.com/zh-cn/solutions/data-lake/>。用户不用关心各个独立的组件。Azure Data Lake 包括了所有所需的功能，使开发人员、数据专家和分析师可以更轻松地存储任何大小、形状和速度的数据，以及跨平台和语言进行各种类型的处理和分析。它消除了插入和存储所有数据的复杂性，同时启动更快，可与批量、流式、交互式分析一起运行。

这种服务底层是构建在基础的托管集群上的，把各种服务组合在一起，提供统一的访问。服务本身提供自动弹性伸缩的能力，数据自动搬迁，自动保障客户的 SLA。

Azure Data Lake 主要包含两部分：Azure Data Lake Store 和 Azure Data Lake Analytics。

1) Azure Data Lake Store

Azure Data Lake Store 的工作界面如图 11.8 所示。

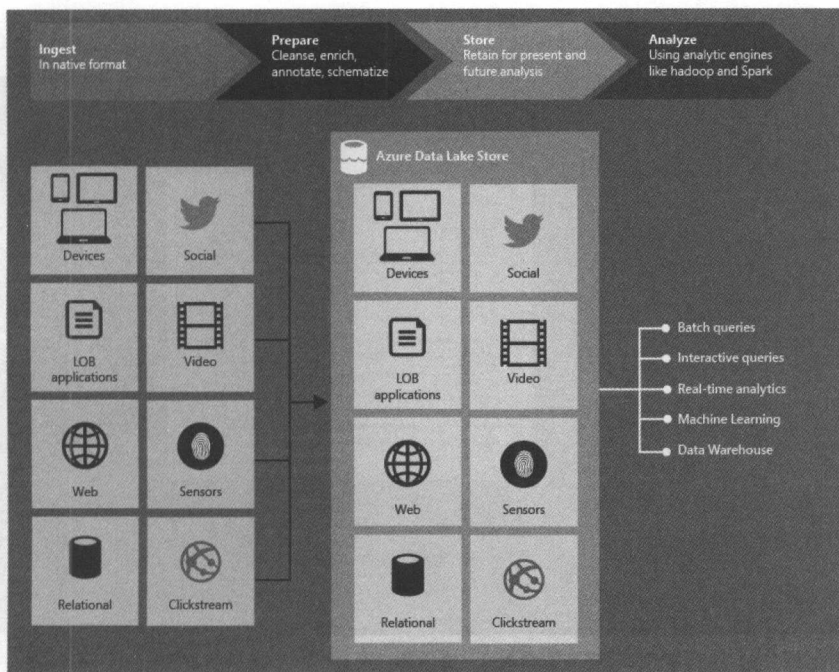


图 11.8

Azure Data Lake Store 是一个专门的文件存储，对外提供 HDFS 兼容的 REST APIs Web HDFS。Hadoop 的分析工具如 MapReduce/Hive 可以直接访问和分析存储在 Data Lake Store 上的数据。所有数据都可以以原始格式存储，支持结构化、非结构化和半结构化数据的存储。可以通过 SDKs、Azure Portal、Azure Powershell 等多种方式对 Data Lake Store 进行操作。

其和 Amazon S3 的最大区别是不限制文件大小。Amazon S3 将数据作为对象存储在称为“存储桶”的资源中，对象大小最多可为 5TB。而 Data Lake Store 对单个文件的大小没有限制。

在性能上，通过把文件分布在不同的存储服务器上，通过并发来提高吞吐量，以分析场景下对吞吐量的性能要求。

2) Azure Data Lake Analytics

Azure Data Lake Analytics 的工作界面如图 11.9 所示。

Data Lake Analytics 的关键特点如下：

- 提供 U-SQL 统一语言，集成了 SQL 和 C# 编程的能力。
- 支持批处理、流式和交互式分析。
- 集成了 Visual Studio 工具、更好的编程和 Debug 代码。
- 构建在 YARN 之上，资源根据任务的需要自动进行弹性伸缩。它根据客户的任务进行计费，有别于集群托管模式。

Azure Data Lake

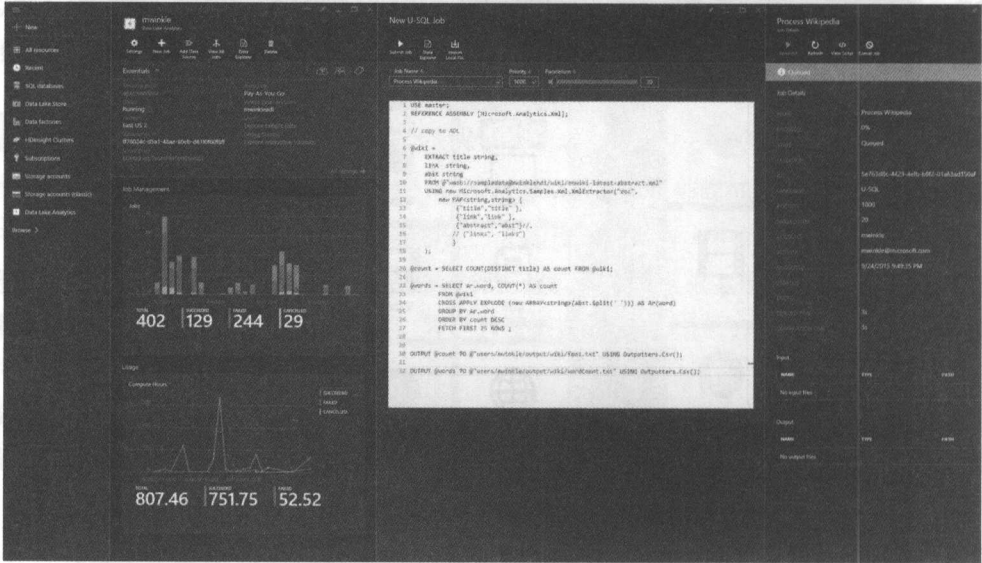


图 11.9

3. 服务模式的优缺点

服务模式是一种更高阶的使用模式，相比集群托管模式大大降低了大数据的使用门槛。但是每家服务模式的服务没有标准，客户使用这种方式有被供应商绑定的风险。

11.4 小结

本章主要介绍了两个方面的内容；一个是应用上云，另一个是大数据上云。大数据作为基础设施，和普通的应用上云有着相同与不同之处。相同的是都是服务化，利用云资源的弹性、免运维等能力。不同的是大数据作为基础设施，对性能、可靠性等要求相比应用做微服务服务化改造的难度更大。从目前来看，微服务只适合应用。

本章讨论的都是组件的服务化，这只是其中一种模式，按 IaaS/PaaS/SaaS 可以理解为 PaaS 服务。业界还有另外一种趋势，即大数据 SaaS 化，进一步将组件、算法、数据结合起来的综合能力封装成 API，或封装成可用的 SaaS 软件等。大数据 SaaS 化未来会有广阔的空间。

第12章

大数据技术开发生态

第三部分

大数据文化

第 12 章

大数据技术开发文化

本章不具体讲哪个组件的开发技术，只想和大家一起探讨一下开源文化、理念，以及大数据开发模式的一些改变。

12.1 开源文化

软件开发领域，开源源远流长，大数据更是一出生便和开源紧密结合在一起。大数据技术和开源为什么联系得如此紧密，笔者认为有两个主要原因：

(1) 大数据是基础设施，基础设施如 OS，不是简单的一个公司或者个人就可以完全负责开发和维护的，所以通过开源协助降低开发成本是更好的选择。

(2) 基础设施技术先进是一方面，另一方面则是需要应用的支持。所以通过开源构建生态，吸引应用的共同发展，也是促进大数据技术和开源紧密联系的驱动力之一。

讲到开源，肯定要讲到 GitHub。Git 是一个分布式的版本控制系统，最初由 Linus Torvalds 编写，用作 Linux 内核代码的管理。在推出后，Git 在其他项目中也取得了很大成功，尤其是在 Ruby 社区中。目前，包括 Rubinius、Merb 和 Bitcoin 在内的很多知名项目都使用了 Git。Git 同样可以被诸如 Capistrano 和 Vlad the Deployer 这样的部署工具所使用。

GitHub 提供 Web 界面，采用社区运作，托管各种 Git 库。作为开源代码库及版本控制系统，GitHub 目前拥有 140 多万开发者用户。随着越来越多的应用程序转移到云上，GitHub 已经成为管理软件开发及发现已有代码的首选方法。

GitHub 的价值不在于一个版本控制网站，更大的意义在于让社会化编程成为现实，可以认为是程序员的社交网站。

开源意味着更多的人参与进来，对个人和公司而言拥有更多的选择。但是更多的选择并不代表更多的自由；更多的选择导致决策的延迟和满意度的降低；快乐的秘诀在于降低自己的期望值。

12.2 DevOps 理念

DevOps (Development 和 Operations 的组合) 是一组过程、方法与系统的统称，用于促进开发 (应用程序/软件工程)、技术运营和质量保障 (QA) 部门之间的沟通、协作与整合。它的出现是由于软

件行业日益清晰地认识到，为了按时交付软件产品和服务，开发和运营工作必须紧密合作。

12.2.1 Development 和 Operations 的组合

可以把 DevOps 看作开发（软件工程）、技术运营和质量保障（QA）三者的交集。

传统的软件组织将开发、IT 运营和质量保障设为各自分离的部门。在这种环境下如何采用新的开发方法（如敏捷软件开发）是一个重要的课题。按照从前的工作方式，开发和部署不需要 IT 支持或者 QA 深入的、跨部门的支持，却需要极其紧密的多部门协作。然而 DevOps 考虑的不止是软件部署，它是一套针对这几个部门间沟通与协作问题的流程和方法。

DevOps 的引入能对产品交付、测试、功能开发和维护起到意义深远的影响。在缺乏 DevOps 能力的组织中，开发与运营之间存在着信息“鸿沟”。例如，运营人员要求更好的可靠性和安全性，开发人员则希望基础设施响应更快，而业务用户的需求则是更快地将更多的特性发布给最终用户使用。这种信息鸿沟就是最常出现问题的地方。

以下几个因素可能促使一个组织引入 DevOps：

- （1）使用敏捷或其他软件开发过程与方法。
- （2）业务负责人要求加快产品交付的速率。
- （3）虚拟化和云计算基础设施（可能来自内部或外部供应商）日益普遍。
- （4）数据中心自动化技术和配置管理工具的普及。

有一种观点认为，占主导地位的“传统”美国式管理风格（斯隆模型 VS 丰田模型）会导致“烟囱式自动化”，从而造成开发与运营之间的鸿沟，因此需要 DevOps 来克服由此引发的问题。

DevOps 经常被描述为“开发团队与运营团队之间更具协作性、更高效的关系”。由于团队间协作关系的改善，整个组织的效率因此得到提升，伴随频繁变化而来的生产环境的风险也能得到降低。

12.2.2 对应用程序发布的影响

在很多企业中，应用程序发布是一项涉及多个团队、压力很大、风险很高的活动。然而在具备 DevOps 能力的组织中，应用程序发布的风险很低，原因如下。

1. 减少变更范围

与传统的瀑布式开发模型相比，采用敏捷或迭代式开发意味着更频繁的发布、每次发布包含的变化更少。由于部署经常进行，因此，每次部署不会对生产系统造成巨大影响，应用程序会以平滑的速率逐渐生长。

2. 加强发布协调

靠强有力的发布协调人来弥合开发与运营之间的技能鸿沟和沟通鸿沟；采用电子数据表、电话会议、即时消息、企业门户（如 Wiki、SharePoint）等协作工具来确保所有相关人员理解变更的内容并全力合作。

3. 自动化

强大的部署自动化手段可以确保部署任务的可重复性、减少部署出错的可能性。

12.2.3 遇到的问题

很多组织将开发和系统管理划分成不同的部门。开发部门的驱动力通常是“频繁交付新特性”，而运营部门则更关注 IT 服务的可靠性和 IT 成本投入的效率。二者目标的不匹配，就在开发与运营部门之间造成了鸿沟，从而减慢了 IT 交付业务价值的速度。

更小、更频繁的变更意味着更少的风险，让开发人员更多地控制生产环境，更多地以应用程序为中心来理解基础设施，定义简洁明了的流程，尽可能地自动化，促成开发与运营的协作。

一般而言，当企业希望将原本笨重的开发与运营之间的工作移交过程变得流畅无碍时，他们通常会遇到以下三类问题。

(1) 发布管理问题：很多企业都有发布管理问题。他们需要更好的发布计划，而不止是一份共享的电子数据表。他们需要清晰了解发布的风险、依赖、各阶段的入口条件，并确保各个角色遵守既定流程行事。

(2) 发布/部署协调问题：有发布/部署协调问题的团队需要关注发布/部署过程中的执行。他们需要更好地跟踪发布状态，更快地将问题上升，严格执行流程控制和细粒度的报表。

(3) 发布/部署自动化问题：这些企业通常有一些自动化工具，但他们还需要以更灵活的方式来管理和驱动自动化工作，而不必将所有手工操作都在命令行中加以自动化。理想情况下，自动化工具应该能够在非生产环境下由非运营人员使用。

要开始优化发布流程，可以从问题识别开始：看看上面提到的哪种问题在你的团队中具有最高的优先级。

12.2.4 协调人

这是企业级 IT 组织中一个新出现的角色，其主要任务就是协调安排将企业级软件部署到预生产环境。对发布协调人的需求来自以下几方面的原因。

(1) 需要弥合开发与运营之间的鸿沟。

(2) 基础设施日益变得复杂：为了运营 Web 应用，需要多层基础设施和多种平台。

(3) 发布频率上升（由于敏捷和迭代式开发的引入）。

(4) 分布式团队：位于全球多个地点的、包含外包人员的、混合开发/测试/基础设施的团队。

发布协调人的角色（也被称为部署协调人或集成协调人）源自发布管理或发布工程团队。这个角色与航空交通管制有些类似：实时协调不同团队的行动，有效使用共享的资源（空域、航道、跑道、航站楼），达到组织的总体目标（安全起降）。

传统意义上的发布管理往往只关注软件变更的计划与管理，而发布协调则需要控制“将特定软

件变更发布至生产环境”的整个过程。这项工作需要系统地管理所有与“将代码构建并部署到生产环境”相关的技术任务，也被称为“发布工程”。

变更管理是跟踪企业 IT 环境中各种变化（不管是应用程序还是基础设施的变化）的基本原则。变更管理是 ITIL V3 的核心之一。

12.2.5 成功的关键

1. 文化冲击

平稳的文化过渡是让 DevOps 获得长期成功应用和增强发布软件产品的综合能力的关键。第一步是明确 DevOps 的定义，调动开发和运营部门之间的协作，鼓励运营人员采纳软件开发方法，并利用云计算基础设施来完成真实的测试和代码部署。

在软件开发、测试、质量保证（QA）、集成、预生产和生产部署等方面的任何旧小团队必须打散，因为每个小团队都可能拖延开发周期，并且带来不可预料的问题。

以上策略能更好地整合开发和运营人员，通过整合团队成员来产生效益。例如，在讨论运营解决方案或扰乱事后评估报告时应该邀请开发人员加入。相反，应该邀请运营人员列席开发人员规划会议。让交叉组合的工作模式成为制度，可以让团队之间合作融洽，消除沟通不畅导致的延误或疏忽，使 DevOps 的推进更加有效。

这种文化上的改革并不容易，它需要公司提供统一的考核标准，以相同的形式衡量开发人员和运维人员的业绩；培养一种团队精神，让大家一起朝着一个共同的目标努力，而不再只是为了从前各自的狭隘的小团体目标。在这里，有时可以运用岗位轮换或者知识共享的方法。

2. 齐全的工具箱

想要超越文化的影响，组织还必须依靠各种 DevOps 工具。例如，开发人员编写代码需要工具，QA 测试人员完成新版软件的部署需要工具，环境准备、将新代码在测试系统和生产系统之间迁移也必须用到云资源调度工具。工具本身都不是问题，重要的是能够让各种工具互相配合，在软件的生命周期内提供支持。

12.3 速度远比你想要的重要

效率高的明显好处是单位时间内能完成更多的工作。但这只是冰山一角，如果工作速度快，你就会倾向于低估做事的成本，因而乐于完成更多的工作。

举个例子，假设你每写一篇博客都要花 6 个月。这样，当周六你宅在家里无所事事时，可能也不会想写博客，因为你觉得这件事做起来太漫长了。

更糟糕的是，因为写博客的进度慢，所以更不愿意坚持。因为学习一件事最好的方法就是一次又一次重复，而这件事情的时间周期太长了。这也就是 ToDoList 通常完成这么慢的原因：我们会对

其产生莫名的厌烦感。如果一直往里面添加拖着不做的事情，总有一天，ToDoList 会被弃用。

如果及时回复别人的邮件，他们就乐于给我们发送更多的邮件。发送者总是渴望得到回复，这种渴望驱使他们写邮件。换句话说，是速度带来了更多邮件，因为发送者心中低估了这种信息交换的成本。他们知道自己所做会得到回应，所以更愿意去做。

现在网络发达了，公认的一件事就是网站响应速度低会流失用户。反应迟钝的网页就像崩溃了一样，它会使用户受挫，或许就是因为用户的行为没能及时得到回报。

Google 搜索响应速度远近闻名。因为它知道，如果搜索响应快，你就会搜索更多。原因就是，它会鼓励你尝试搜索，很快得到反馈，然后你会再去尝试。当你有了一个想法时，搜索不会让你失去这个灵感，你会认为去 Google 搜索的成本近乎于零，它就像你思维的一部分。

职场中也是同样的道理，做事快的员工会被分配到更多工作。道理很简单，人们都有懒惰心理，大家都想保护自己的卡路里。将工作分配给做事慢的员工去做，光是想想就觉得厌倦。当你要分给这种人工作时，脑海里就会浮现出进程被耽误好几天的情形，会不自觉地看见这些人拖延工作的样子。但做事快的人就不一样，他们的时间看起来“很便宜”，你让他们做某些事情的时候，就知道他们很快会做完，马上就可以再分给他们别的事情做，所以你就会更倾向于分给他们更多的任务。很讽刺，不是吗？公司里最有价值的员工却因为做事比较快而要干最多的工作。

总结一下，规则就是：速度快的系统因为吃得快，所以被喂得更多，速度慢的系统会饿死。

再举两个例子。适用于个人的这些道理，同样也适用于组织。如果顾客发现某家裱画框的店每完成一幅需要两个月，那么他们就会去别的店；如果贡献者发现某个组织接受代码很慢，他们就不再乐意贡献代码。反应慢的系统很糟糕，就像长满了青苔的建筑物，死气沉沉。人们都喜欢有生机的东西，喜欢反馈及时的系统。

开始做一件事的动力，一部分来自对工作画面的想象。一般真正做的时候不会像想象得那么难。但如果想象中成本很高，做起来是个苦差事，就需要下更大的决心才能开始。

“慢”就是这幅画面中重要的成本之一，时间无价。所以当我们认为某项工作很慢时，就会潜移默化地为其添加额外成本。每次想到这种工作，就会情不自禁地想去拖延。

这就是速度为什么重要的原因。

因此，对于要重复做很多次而且必须做好的事情（例如写作、修复 Bug），就应该尽量做快一些。

这并非建议马虎行事。督促自己比平常做快一些是好事，因为在你心里，这将花费更少的时间，也更容易迈出开始脚步，你能完成的工作将会更多。在做更多的同时，质量也会更好（只要你认真），最终达到又快又好的效果。

做事快很有趣。如果你是一个快笔头的写手，就可以经常试验“新”想法，而不会陷入某个泥潭中挣扎很久。你的 ToDoList 可以很快划掉，更乐意往里面添加新东西。随着你不断完成更多的稿子，整个工作充满活力。你会感觉自己聪明能干、脚踏实地。当有大任务出现时，你就敢挺身而出。

你会看到在大数据领域流行的两种语言 Scala 和 Python，提高开发速度和更优雅是它们的目标之一，也使它们更流行。Scala 集成了函数编程和对象编程的优点，使程序语言处理数据变得更简单。

Python 一贯的简单易懂的风格，再加上集成了大量的机器学习的数据库，让 Python 机器学习领域再度流行起来。

12.4 小结

最后再总结一下，大数据不是一个技术名词，而更多的是一种理念，是改变传统技术和思维的理念。

正是新技术、新思路的引入，使社会不可阻挡地进行着变革。在历史的洪流中，只有开放和学习才能跟上时代的步伐。

出版编辑联络：安 娜

微信&QQ：80303489

邮箱：anna@phei.com.cn

大数据架构详解

从数据学习到深度获取



博文视点Broadview



@博文视点Broadview

上架建议：数据架构

ISBN 978-7-121-30000-4



9 787121 300004 >

定价：69.00元



策划编辑：安娜
责任编辑：徐津平
封面设计：侯士卿